MASTER IN COMPUTER SCIENCE

Hamsters
a new task model for interactive systems

Ben Amor, Mohamed

Award date:
2009

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique

Année académique 2008-2009

Hamsters–A New Task Model for Interactive Systems

Author:
Mohamed Ben Amor

Supervisor:
Prof. Monique Noirhomme

Mémoire présenté en vue de l'obtention
du grade de maître en informatique.

Septembre 2009

Abstract

Task Analysis is considered to be one of the most powerful methods available in Human-Computer Interaction discipline. To support this important method we developed a new Task Model named *Hamsters*. It allows the specification of relevant information concerning different tasks related to the system in a formal way enabling analysts to use them in a systematic fashion. To achieve this we studied existing task models, selected important and successful concepts to preserve, and introduced various improvements at different levels: meta-model, notation, simulation and implementation. Hamsters is a very flexible model because it was developed with two principles in mind: *modularity* to support extensions; making it adaptable to various system types (critical safety systems in our case), and *openness* to other models (to complement them and enable cross-verification).

Résumé

L'analyse de tâches est considérée comme l'une des activités les plus utiles dans le domaine d'Interaction Homme-Machine. Afin de modéliser cette importante méthode, nous avons développé notre propre modèle de tâches avec comme nom *Hamsters.* Il sert à encoder des informations pertinentes aux différentes tâches liées à un système d'une manière formelle permettent aux analystes de les utiliser après d'une façon systématique. Pour y parvenir, nous avons étudié des modèles de tâches existants, sélectionné les concepts importants à reutiliser, et introduit des multiples améliorations aux différents niveaux: méta-modèle, la notation, la simulation et l'implémentation. Hamsters est un modèle très souple grâce à sa conception qui se base sur deux principes: la *modularité* afin d'accepter plusieurs extensions de différents types de systèmes (*systèmes critiques* dans notre cas), et l'*ouverture* à d'autres modèles (pour les compléter et permettre à les *cross*-vérifier).

Acknowledgment

This master thesis would not have been possible without the help of some persons that I feel so grateful to them for all the contribution and support they provided me.

In particular, I would like to thank my thesis supervisor Mrs. Monique NOIRHOMME for offering this interesting subject of research. I am grateful to every remark and detail she gave while reviewing my thesis.

I want to express my highest gratitude to the team of IHCS (Interacting Humans with Computing Systems) from the IRIT, *Institut de Recherche en Informatique de Toulouse*, that unless their continuous support and hospitality, I would not be able to pursue this work. Especially I would like to thank Mr. Philippe PALANQUE, the chief of IHCS team and my internship mentor, for all the insights, ideas and references he provided me, despite his enormous commitments and busy schedule. Similarly, I want to show high gratitude to Mr. David NAVARRE who has provided assistance at almost every level of this work with his contributions, feedback and highly constructive critical views. Not to forget also both Mr. Eric BARBONI and Mr. Jean-François LADRY for their highly appreciated help during my stay in Toulouse and for being such an outstanding work mates. I would like also to thank Mr. Marco WINCKLER for all his help and support.

Some parts of this document were not possible without the help of three master students from the University of Toulouse Paul Sabatier: Fabien ANDRE, Yousssef AZZOUZI and Raphael HOARAU. I would like to thank them so much for their cooperation.

I am thankful also to all my professors at the University of Namur who allowed me to get some deep insights into computer science and provided me with fundamental knowledge that enabled me to carry this research and relate it to the different subjects we studied. In particular, I would like to thank Mr. Vincent ENGLEBERT for his remarks about my implementation chapter.

Finally, I will never forget to mention my family which despite the distance, has kept supporting me at all times. I dedicate my thesis to them for all what they did for me.

Contents

List of Figures

List of Tables

Context and Introduction

Context

The impetus for this research comes from two separate contexts. The first is about a wider context which is *Task Modeling* for Human-Computer Interaction and the second is a more practical one within a research project named *Tortuga*.

During, the last two decades, the interest in model-based approaches in Human-Computer Interaction (HCI) has been growing. Different models and approaches were proposed to support various purposes in HCI. Among, these models we distinguish two major types: System-Oriented and User-Oriented. The first is similar to most models found in software engineering but gives more weight to interaction and usability in its constructs. The second serves to model everything related to the user and useful to create better designs. Those models are essential because of the intake they provide to support *User-Centered Design*. The most recognized model of the latter type is Task Modeling aiming at supporting Task Analysis in a more systematic and formal way. Among the HCI community, "*task analysis is potentially the most powerful method available to those working in HCI and has applications at all stages of system development, from early requirements specification through to final system evaluation*" [Diaper 1989].

The second more practical context which is the Tortuga project. The project is financed by the CNES (*Centre National d'Etudes Spatiales*; the French Space Agency) and focuses on standardizing the "*automation*" service specifications written by CCSDS (The Consultative Committee for Space Data Systems) working group. The research intends to assure an improved operability (reliability, efficiency, error tolerance...) of ground segment applications using model-based user-centered design approaches. In particular the project aims at defining various methods and models to represent the complex socio-technical system from different perspectives and more importantly to support *cross-verification* of these models. The cross-verification assures that all models are coherent and compatible. This coherency at the model level will guarantee in turn the system coherency: monitoring and control interface, operator tasks, training material... Our research in this thesis relates to the first phase: models definition. Precisely, we will work on defining a new *Task Model*.

Introduction

With the help of the rapid evolution that touched every aspect of our life during the last and current century, our life seems to become much easier especially with the introduction of new solutions and technologies. However, this development brought with it at the same time more complications to everyday situations, especially work environments. It was clear that such complications will have direct implications not on tools but on the human performance. Such complications can be traced to different factors of work situations from organizational hierarchies to state of the art tools employed. When it comes to tools, the problem has its root in the way we view *systems*. Systems are not only a set of technological artifacts, they have an environment and more importantly a *human element* (people) to whom we designed the system at the first place. The solution to this problem should not simply ask us to change our views but the whole methodology we employ while designing systems. The most known methodology that promises to offer this prospect is *User-Centered Design* (UCD) which as it name suggests puts the user at the heart of the system design.

Looking at users as a central part of the design requires from us to define some formal approaches to integrate details related to them. In particular, as stated above, we need to capture information on various factors that influence their performance when interacting with the system. Therefore, we need to develop methods and techniques which could help to understand and analyze user situations in order to enhance Human Performance (HP). It is obvious that the best way to have a better HP is to analyze human performance itself. Thus, this analysis has to underlay a process of collecting, classifying and interpreting data related to different tasks he or she performs. This process is known in the literature as *Task Analysis* (TA). However, how to relate this analysis to the system design and use it in a systematic way requires defining a method to *model* our analysis. This is the *raison d'être* of *Task Modeling*, providing formal descriptions of user tasks for various systematic uses. Since its inception as an important research topic in HCI, multiple model definitions and tools were developed to support Task Modeling. However, these models are still seen as research subjects and consequently do not enjoy wide adoption from the software industry. This is due to their diversity and high level definitions. Actually, their uses and applications depend on various factors mainly the foundation of employed task analysis itself and the awaited purposes of it. So in an attempt to solve these pitfalls, we aim in this research to define a new task model that builds on existing ones while trying to avoid past problems. The definition of this task model itself will take into consideration various factors that are regularly omitted ranging from higher-level concepts such as abstraction and modularity to more specific concerns such as notation and interactivity. In addition, our task model aims to be *extensible* and *open*. Extendability allows domain specific aspects to be encoded inside the model easily (e.g. aspects related to safety critical systems). While, openness makes our model accessible to other models and tools, enabling complementarity (models of the same system from different perspectives) and coherency (models cross-verification).

Outline

The remainder of this thesis is organized as follows:

Part 1 It gives a brief literature review on topics related to our research. The first chapter shows the importance of User-Centered Design and model-based approaches in developing interactive systems in particular and software in general. The second chapter presents the state of the art in task modeling. The third chapter attempts to extend the previous one by analyzing some selected existing models and concludes by providing taxonomy for task modeling.

Part 2 It presents our research contributions and describes our task model– Hamsters (stands for *Human-centered Assessment and Modeling to support Task Engineering for Resilient Systems*) from various angles. The fourth chapter discusses the foundation of our model. The fifth chapter carries on the former one by presenting the meta-model of Hamsters in detail. The sixth chapter presents our notation language. Finally, the seventh chapter provides details related to our implementation.

Part I

Literature review

Chapter 1

Towards a UCD Model-based approach

1.1 Introduction and Context

Following the famous software crisis, various efforts were put by different entities from the academic world to governments aiming to find a way out of this chaos by making the Software Engineering discipline *more disciplined*. Most researches focused on finding the best software lifecycle or methodology that can lead a project to success. In parallel to this research, there was also another tendency to create and provide better support tools and development approaches to deliver better software products; mainly through computer-assisted software engineering tools (CASE tools).

Different solutions were proposed but when confronted with real-life uses they tend to continue posing problems, although mostly performing better than former approaches. We are not going to discuss these different methodologies and approaches or evaluate them. We will focus on the major weaknesses found to be the source of "trouble" for the software industry.

The most important factor of failure of any software products is *wrong requirements* (the building blocks of software) [Brooks 1987; Filho and Kochan 2001] . It is logical that if they have collected wrong requirements they will not provide the awaited product (requirements being the first process in software lifecycle). The second most important factor is *change*. Along the software development lifecycle, stakeholders will mostly impose or introduce changes to the requirements or to the in-development product (even prototypes). At first, wrong requirements and constant-change seem to be not connected but actually they have the same root-problem: misunderstanding. It is agreed both in the academic and industrial world that improving understanding between the stakeholders (including engineers) is a key-solution to major problems in Software Engineering. We would like to make clear that by *understanding* we do not mean only communication but also understanding the factors that will surround the software usage. Among stakeholders the *user* is considered the key-success factor. This claim has been demonstrated again and again by the Standish Group reports; User involvement is the number one success factor and reversibly lack of user involvement is the number one failure factor. These two major mentioned factors above (requirements and change) provide us with clues that lead us to two important key-concepts: *User* and *Process-automation*.

The first concept *user* represents any person who uses or can be affected by the use of the system, more generally some employ the term *human* instead. Consequently, if we want to have better requirements and understanding of the system we need to understand the user. This is especially true for interactive-systems where the user interaction plays a key-role in the software usage, nevertheless remains important also for other software system types. This new methodology or software philosophy is mostly known today under the name *User-Centered Design* (UCD) and sometimes called *Human-Centered Design* although some scholars such as in [Gasson 2003] emphasizes on the differences between a Human-Centric perspective and that of a User-Centric one.

The second keyword *process-automation* is about providing better traceability enabling an automated back and forth navigation along different software processes. Traceability is considered the viable answer to the *change* problem. If we have an automated software-process, we can integrate any change that will be reflected in all phases from requirements to implementation. In our case, we will discuss the most known proposed solution promising to provide this functionality: *Model-Driven Engineering*.

To better understand the advantages of employing new approaches, we will start by demonstrating the pitfalls of current, or better call them "traditional", ones. You will notice that our claims apply to a wider circle of systems (not only software ones). However, sometimes we will try to be very specific by focusing on a special type of software called *Critical-Interactive Systems* (CIS) which forms the basis of our showcases. Next, we will talk in more details about the User-Centered Design and later present Model-based approaches. Finally, we will demonstrate how a hybrid design mixing both approaches is very beneficial mainly for interactive systems.

1.2 Current approaches (mainly TCD)

Most previous and current software approaches tend to look at the software problem from an angle different than that of the user. At first, the goal was developing a product that provides a specific feature. Later on, we find out that the product fails regardless of its relative correctness in terms of its defined features or goals.

Traditionally, most systems were engineered from a self- and/or technology-centered perspectives. The first type is simply a design where engineers look at the system from their perspective. In other words, they develop a system based on their needs and wants; even if they are not related to the system usage in any way (i.e. they are not and will not be end-users themselves for instance). Bruce Tognazzini pointed out that even when they tend to look for another perspective they usually ask people like themselves[Tognazzini 1996, p. 230]. The second type is considered more dangerous than the first one. In this case the human factor has almost no role in developing the system: technology is taking the lead in the system design.

In the following sections, we will detail the aspects of current non-user focused approaches mainly the Technology-Centered one. Next, we will list the different consequences that such approaches result in.

1.2.1 Aspects

The problem of *Technology-Centered Design* (TCD) does not only concern Software Engineering but all engineering disciplines providing products powered by technology. We should note that in most cases building a system from a technological perspective is not an explicit design choice taken by the developers. The problem of TCD is not new, different scholars discussed the dangers associated with it from early days. These views were not only discussing the implication of TCD on its ability to deliver a successful product, but further how TCD is affecting our own way of living. Particularly, Henry Dreyfuss was very critical of how technology is starting to drive our lives in his book *Designing for People*. He said that "Somehow, we must find again our sense of individual values, lost in this century of enormous technological advance". Other scholars as we will show below were critical of how it is a primary cause of design failures, further how it can be an ordeal rather than a source of relieve.

When we analyze most tools that were designed or built from an engineer and/or technology perspectives, we identify various problems especially related to the usability of the system. To be more specific, we can cite an example taken from Aircraft Systems. In the beginning, engineers installed a display for each system to tell the operator or pilot how a specific item is performing (engine temperature, coordinates, altitude, speed...). At first the number of indicators was small but with the rapid development of the aviation industry more and more displays were added. This increase of indicators has no doubt provided us with additional data but at the same time started to make operators get confused as a result of the increasing number of displays they have to deal with. To face this critical situation, multipage displays were introduced but still not able to solve the issue. In fact, the number of pages as also the quantity of displayed data continued to grow exponentially[Sexton 1988]. The problem is not simply reducing the number of displays, what the operator is facing in this kind of situations is reacting to changing environment based on information provided by these indicators. The challenge lies on how he or she would find, sort, integrate and react to this overwhelming quantity of data (especially to identify the necessary information out of the data sea). This kind of situation is an example of *Information Gap* (see figure 1.1 on the following page). In other words, the technology is forcing the operator to adapt to this extremely critical situation and give him/her the whole responsibility of sorting and locating the required information. This explains why the job of an operator requires some special skills not only due to its criticality but also the required extra-abilities to cope with complex technology-centered systems. Actually this pushes us to ask an interesting question: How much time operators are spending to adapt themselves to these systems during their studies compared to that of actual learning?

What developers and engineers miss in this kind of situations is not taking the limitations of the human processing system into account. Unfortunately, we have a limited short-term memory and certain bottlenecks in our abilities of information processing. What these displays are doing is conveying scattered raw data to operators. This type of design is not well suited for human tasks because it shows data in a distributed often unrelated way following a technology-centric view. In addition, it provides Information from a discipline-centric view (think

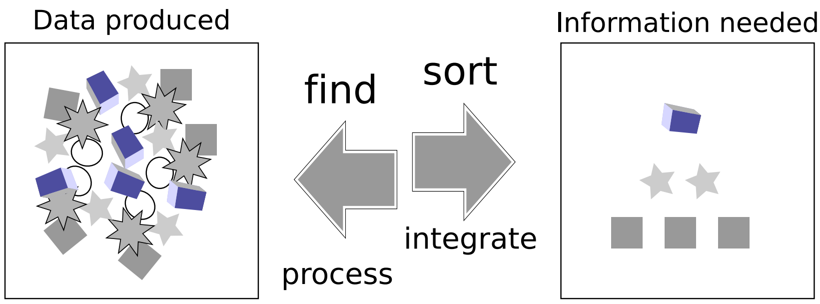


Figure 1.1: Information Gap [Jones and Endsley 2000]

of using and IP address instead of a friendly easy-to-remember domain name in Networking). As mentioned above, an extra mental processing will be needed in order for the operator to select the relevant information, let alone taking the right decision later. This required processing overload can be considered as a major source of [disastrous] errors.

To demonstrate our last claim, we will present some data collected from our main example in aviation. Most of these data were collected from reports of past incidents which are usually considered the best source for probable causes [C.W 2000]. These causes can either be traced to engineering faults or human-errors. Statistics show that the percentage of accidents caused by engineering failures was reduced significantly, but on the other hand failures attributed to humans are on the rise. In fact, the operator is considered a causal factor in between 60% to 85% of all accidents [Nagel 1988; Sanders and McCormick 1992]. If we look beyond aviation, we would conclude that this problem can be identified in other sectors. For example, the National Academy of Science in the United States of America has estimated over 44,000 deaths annually which are *only* attributed to human-errors in the medical field, two thirds of which are considered preventable [Kohn et al. 2000]. It is apparent that these errors are human in nature but we cannot claim that humans are faulty or they are doing this intentionally! The real cause is the technology-centered designs which are mostly, if not always, ill-suited for humans. These designs push our human performance beyond its capabilities increasing at the same time our confusion and more importantly our chances of committing fatal errors. That is why the more accurate term *design-induced* was coined by experts to denote this kind of errors instead of the misleading human-error. For the rest of the document, both terms will be used interchangeably.

Early researches that tried to find a solution for this problem focused on one solution which is decreasing the human intervention. In other words designing systems where humans interaction with the system is reduced to the minimal, thus logically reducing errors. Obviously, this was mainly done by a significant increase of automation. The idea was: making most tasks automated and ran by the system will reduce at least the workload. Unfortunately, the increase of automation was not effective as perceived at first. To demonstrate this we can analyze a use case from Boeing. In the period between 1959 and 1981, Boeing reported that in 76% of hull loss accidents, the flight crew were a primary factor. Between 1982 and 1991, more planes with increased automation were flying in the skies but the percentage was reduced only to 71.1%. Actually, while we think only of the utility that came with these automated systems, we forget that introducing new systems means new unexpected errors. These errors are basically induced by the increase of system complexities, loss of situation awareness, system bitterness and workload increases at inopportune times[Billings 1997; Parasuraman et al. 1996].

1.2.2 Consequences

In this section, we will focus on the consequences of following a technology- and/or self-centered-design. Precisely, we will take a closer look at the major problems that are often the result of these design approaches.

Systems are designed in the first place as tools empowered by technology to make human tasks easier. This means that they are supposed to make our life easier and not the opposite. Alas, reality is different. In fact humans are no longer employing technology but trying to adapt to it; if not sometimes resist or compete with it. It is clear that a set of minimum skills and knowledge are required to use these systems but spending most of the time adapting and learning is a sign of bad design. Actually, users are no longer concerned about how this system can help them perform their tasks in an easier and efficient way, but instead they are shifting their attention to how to use the system. So instead of solving the existing problem, the system created a new one. This shift can have fatal impacts especially on critical-interactive systems. Similarly, it can affect other types of software such as pushing users away from a website because they are not ready or sure of their skills to manipulate its services (this is especially important for e-commerce websites).

Technology tools today are known to have an enormous processing and memory power. Simply integrating these tools into supporting human tasks can be very dangerous. This processing power exceeds ours. The technology-centered systems push our processing abilities to the extreme raising very dangerous concerns about our decision-making process. Logically, this disturbance will result into committing different types of errors which are going to be attributed wrongly to humans later.

Another important weakness of TCD is producing sub-optimized systems. Delivered systems are usually complex, contain tons of data and difficult to interact with. All these leaks would always produce a non-user friendly and even non-usable system. In this case, the system has two major flows:

1. It is not optimized to perform the awaited tasks as it does not really help us in sorting and locating the information: engineering flows.

2. It is supposed to improve and optimize the working environment but the opposite happens and it became a burden: impact on its external environment.

Before closing this section, we can summarize the problem of TCD and SCD in the following main issues:

- Tools are built from an Engineer/Designer's perspective.

- It is up to the user to adapt himself or herself to technology.

- Users adapt their environment to accommodate what has been designed.
- Users are more concerned about how to use the system instead of focusing on their tasks.
- Users alter or use the design in an unexpected or unusual ways from what designers intended in the first place.
- It pushes the human processing abilities to the extreme in complex systems.
- Increase the probability of committing errors.
- The developed system is not optimized (i.e. not usable, not user-friendly).

1.3 User-Centered Design

As demonstrated above, we often create bad designs following traditional approaches. This highlights clearly the important gap that shows up later between how the developers intend for the technology to be used and its actual use through analyzing users behaviors. Most researches knew that the missing link is the user. In other word we need to reposition our view of the user when designing. The user shall be no longer just a client or a customer who is involved only at the start and at the end of the development process. To face these issues, a new design philosophy or approach emerged called the *User-Centered Design* (UCD). We used the term philosophy to show that it is not a rigid neither a mature concept ; for instance it does not have a well-defined and all-agreed methodology. We can arrive to this conclusion by just looking at the different definitions and methods of UCD found in the literature and industry. Despite this, practitioners and researchers continue to advocate User-Centered Design which enables people to reach their goals while taking into account the natural human limitations, and produces generally more intuitive, efficient and pleasurable to use systems [Sharp et al. 2007].

We can find the earliest reference to UCD in [Gould and Lewis 1983] where they mentioned some UCD principles like continuous contact with users, usability criteria and evaluation and iterative design. However the official launch of the term UCD is credited to the seminal work of [Norman and Draper 1986]. In his book "*The Design of Everyday Things*", Norman approached the good/bad design problem from a psychological perspective. He coined the term User-Centered Design to describe any design based on user needs. A good design according to Norman should make the user (1) figure out what to do and (2) allow him or her to know what is going on. For him, UCD involves simplifying difficult tasks, making things visible, make it easy to evaluate the current state of the system, follow a natural mapping (between intentions and actions, actions and impact, etc.) [Norman 2002].

While the goals of UCD are clear and relevant, how to achieve these goals remains a mystery. This is especially true when it comes to the definition. From the beginning there were many essays and debates to find a definition for UCD. The problem is particularly keen in the HCI community. No doubt that it was an approach that everyone subscribed to, and endorsed, but for which there seemed to be no agreed-upon definition. During CHI'96, the panelist Dennis Wixon pointed out the importance of this problem. If we cannot define what UCD is, then we are faced with allowing virtually anything to be called a "UCD Process." Is UCD a term that describes anything that usability specialists do or is it a specific set of techniques drawn from a larger set of activities that may be a part of system design? Can (or must) we tolerate ambiguity in the definition, or is a precise definition of UCD necessary?

The situation in practice was worse. In the industry each organization has its own version or understanding of UCD. This "diversity" is due to the *high-level* definitions given mostly by the academic world. As Martin Rantzer from Sony-Ericsson suggested, these definitions are too high-level for many organizations. While it is true that organizations usually appreciate this flexibility, it should not be too high so no one can really put it into practice.

The *Usability Professionals' Association* [UPA 2009] defines UCD as "*an approach to design that grounds the process in information about the people who will use the product. UCD processes focus on users through the planning, design and development of a product.*" We are not going to dig further into UCD definitions but the previous points still highlight the important corner-stones. In the next section, we will give an overview of some major UCD principles. Later, we will further detail the goals of UCD and more importantly demonstrate its effectiveness.

1.3.1 Principles

Since the introduction of UCD, different principles and frameworks were built for it. Among the most important we can cite the ISO Standard 13407, *Human-centered design processes for interactive systems* [ISO/IEC 1999]. However, we are going to limit this section to the Gulliksen's framework which outlines 12 principles for successful user-centered system design [Gulliksen et al. 2005]. The reason for this limitation is the fact that all these points were based on extensive look on existing researches (including ISO 13407) and real-world practices. We will add additional references to original works where applicable.

User focus The user should be the major focus. User goals and the tasks needed to rich these goals should guide early the development [Gould et al. 1997; ISO/IEC 1999].

Active user involvement Users should be active in the development process from early stages. Users are no longer "customers" but are involved and participate actively in the development [Gould et al. 1997; ISO/IEC 1999; Nielsen 1993].

Evolutionary systems development This is basically saying we should use both an iterative and incremental development process [Boehm 1988; Gould et al. 1997].

Simple design representations Design representations and terminology should be simple and more importantly easy to understand and grasp by users [Kyng 1995].

Prototyping Early use of prototypes is encouraged to visualize and analyze/evaluate design ideas and decisions. Prototyping here ranges from sketches to small applications; depends on the project and timing [Gould et al. 1997; Nielsen 1993].

Evaluate use in context Usability goals and specific design criteria should be specified so we can evaluate the design against them in cooperation with users in context [Gould et al. 1997; Nielsen 1993].

Explicit and conscious design activities The development process should contain dedicated design activities. For instance User Interface Design and Interaction Design activities [Cooper 1999].

A professional attitude UCD is basically a multidisciplinary approach, thus it should be performed by professional people from different disciplines but with a multidisciplinary cooperation [ISO/IEC 1999].

Usability champion Usability experts should be involved in the development process since the very early stages and through the development lifecycle with clear authority on usability issues [Kapor 1991].

Holistic design All aspects that could affect the future use of the system should be developed in parallel. This resembles to the idea that software does not exist in isolation [Gould et al. 1997].

Processes customization Every UCD process should be adapted so it has a local implementation depending on the organization where it is employed.

A user-centered attitude should always be established The UCD attitude should not be a concern for usability people only. All project members should be committed to the importance of this attitude and the importance of usability.

You might already notice that there seem to be a near consensus on the importance of UCD and its major principles. However, things are not that quite straightforward, as real-world practices pose new challenges. Despite this, UCD has shown that it is capable to achieve most of its goals and demonstrated its effectiveness.

1.3.2 Goals and effectiveness

The main goal of UCD is helping to create good design. A good design in turn provides the following advantages:

1. Reduces human-error.
2. Improves productivity.
3. Wins user acceptance and satisfaction.
4. Improves overall system's usability.

By usability here, we do not mean the traditional definition of the term where focus is given only to interaction but the overall usability of the system (not only the appearance but also include other elements such as filtering and sorting information, in other terms provide users with information they need especially in complex-systems where a huge amount of information is processed).

Measure	Frequency
External (customer) satisfaction	33
Enhanced ease of use	20
Impact on sales	19
Reduced helpdesk calls	18
Prerelease user testing/feedback	16
External (customer) critical feedback	15
Error/success rate in user testing	14
Users' ability to complete required tasks	10
Internal (company) critical feedback	6
Savings in development time/costs	5

Table 1.1: Top 10 cited measures of UCD effectiveness.

To evaluate UCD effectiveness, different surveys and studies were conducted. At first these studies were critical and showed that many UCD-methods in the literature were found ineffective or impractical for a variety of reasons [Gould et al. 1991; Vredenburg 1996]. More recent studies continue to be critical of the current situation especially the lack of clear standards, but at the same time proves that adopting a UCD approach can increase the project success prospects significantly.

To demonstrate the effectiveness we will use data from one important relatively recent survey by Vredenburg et al. [2002]. According to the results of this survey, regarding the perceived impact of UCD, 72% of the respondents reported that UCD methods had made a significant impact on product development in their organizations, by indicating five or higher on a seven-point scale. The overwhelming majority said UCD methods had improved the usefulness and usability of products developed in their organizations, 79% and 82% respectively. "Clearly, there was a consensus that UCD had made a difference". When it comes to the method of evaluation or indicators of UCD impact on the project success, respondents gave different evaluation factors. Table 1.1 lists the most important measures cited by respondents sorted by frequency of occurring.

We will not dig further into demonstrating UCD positive impact on design. You could refer to the cited survey above [Vredenburg et al. 2002] for additional data and statistics related to UCD effectiveness. Also, a book with the title *"User-Centered Design Stories"* by *Righi and James [2007]*, presents and details different real-world case studies on the impact of UCD.

1.4 Model-based approach

1.4.1 Models and Modeling

Models were always an integral part of the human experience. May be we are not aware of their explicit usage but we can find them everywhere. A model is a reflection of something real. For instance, the human creates models of the world with information provided from our five main sensory inputs: visual, auditory, tactile, olfactory, and taste. These models are usually not complete and therefore do not hold a loyal representation of reality. This incomplete representation is primarily the consequence of our limited abilities to capture every detail, and secondly to our subjective view of the world. While the second cause remains problematic, the first is the power force of models: models are not complete yet they are powerful and easy to employ for different purposes. This partial representation makes it easier for us to process and decide based on these models. Thus, we can define a model as a *simplified* representation of a real thing which includes only those aspects of the real-world that are *relevant to the situation at hand*.

The human mind is quite talented in creating and using models. This can be proved from our ability, as infants, to develop sophisticated models of motion, distance, time, and cause/effect in an effort to relate to the new and confusing world around us. Starting from a nearly clean model we try to construct an internal understanding for every new experience. This clearly shows that the idea of creating an *abstraction* of the world in an effort to understand complex situations/ideas is inherent to human way of thinking and reasoning [Lieberman 2006].

What is exceptional about models is their versatility. We can use models in various ways and we keep discovering new ones that continue to demonstrate their usefulness. Among the most important uses of models is communication. If we look at how we communicate we can conclude easily how models are crucial for us. Actually one of the most sophisticated examples of human models is the language. It allows us to express very complex ideas either by using direct mapping (concepts) or by connecting existing abstract concepts to express new ones. Notice by language we are not limited to the spoken one but also other forms of languages. More generally, these internal abstract models we build ourselves are found to be critical in communicating complex concepts and ideas to other people [Mandel 1997; Morgan and Welton 1992]. Although we need to be careful here, those abstractions cannot be used in a universal fashion. The most famous constraint is cultural differences (using two different languages for instance). That is why models are not created in isolation but need to be aligned to a common ground among participants. For example if we would like to explain a new phenomenon, we need to align our model to a shared experience with the intended audience [Schramm 1971]. Therefore, to increase understanding during a communication we need (1) a model that has the right abstraction level and (2) a strong shared experience and/or views on things. To express the second factor, psychologists employ the term *cognitive resonance* to describe a situation of matched views between the modeler and the audience. On the other side, they use the term *cognitive dissonance* to represent a situation where the model is foreign or not close to the audience's experiences and/or expectations.

Abstraction is may be the corner stone of models. Through abstraction we are able to create simple models of complex situations or systems. The resulting simplified representation can be used for communication as mentioned above but in the case of systems for the purpose of reasoning, simulation and analysis. All these benefits and facilities explain why models are becoming more and more frequent in different disciplines, especially young ones. In particular, software modeling and models were seen as a new way to develop systems. Actually models are used both to enhance understanding between different stakeholders and to drive system development. A particular movement known for endorsing the second use of models is the Object Management Group which promotes the MDA (Model-Driven Architecture). Basically the idea is to make the engineering process model-based. Models are no longer a simple representation that help us understand or reason but they can be processed and formalized in a way that they will be able to generate the *system-to-be*. Following this approach, engineers need to model the system using well-defined models, than rely on them to produce final artifacts. What is interesting about this method is the power of automation. In traditional approaches, usually we write the requirements than the design is followed by the implementation, testing, etc. The transition from one software-process to the next is usually done manually. This poses two problems: the additional manual efforts and the lack of traceability. Model-Driven approaches avoid these transitions by introducing the concept of transformations. A transformation is usually a fully-automated operation which transforms one model from one level to another (transformations could be horizontal too; converting one model to another type of model with the same abstraction level). This way, only first models are required to be designed manually by engineers than the next processes will be automated. Not only we achieved a simpler way to describe the system but guaranteed an almost fully automated lifecycle.

1.4.2 Model-based approaches in HCI

In the field of Human-Computer Interaction models were always playing an important role although mostly used for modest purposes at first. In the beginning most researches were focusing on creating models that can design User Interfaces (UI). This movement gave birth to a family of models called *Model-Based User Interface Development Environments (MB-UIDE)*. It aims to define models and develop support tools that can help designing and implementing UIs in a professional and systematic way [da Silva 2001]. It is also interesting to note that HCI has reused many existing models from other disciplines such as cognitive sciences and industrial-management.

When it comes to model uses, HCI was not very advanced compared to Software Engineering (e.g. MDA). Actually most employed models are used to help understand or evaluate the behavior of complex systems. MB-UIDE tried to create fully generative models but in practice they failed to gain a wide acceptance. This failure can be justified by the lack of profound understanding of user's interaction with the system and employing a limited formal declarative languages.

Models in general are located along an axis delimited by two ends:

1.4.2.1 Predictive models

Predictive models, known also as *engineering models* or *performance models* [Mar 1991], are widely employed in various disciplines. They tend to be mathematical in nature. In HCI, they allow the designers to evaluate the human-performance analytically without undertaking resource and time-consuming experiments or prototypes. These models enable us to analyze and evaluate design scenarios without the need to implement the real system which will require additional tools to gather the usage metrics. The most famous examples of predictive models in HCI are the Hick-Hyman Law and Keystroke-Level Model (KLM), used mainly for predicting motor-based behaviors. We will take a closer look at the KLM model in section 2.3.2 on page 22 when discussing GOMS.

1.4.2.2 Descriptive models

Descriptive models are totally different from predictive models. They tend to be metaphorical in nature. Thus, usually they do not yield us to empirical or quantitative measures. However they have different features that make them as powerful as predictive models. Usually this type of models provide us with a description of relevant concepts to the modeled situation. In the case of HCI, it can help us develop frameworks that identify categories or features related to an interface or an interaction. At first, these models seem to be simplistic and of not of direct use but they can help to better understand and design an interaction. As an example we can cite the Key-Action Model which puts keyboard keys into three major classes: (1) symbol keys, (2) executive keys and (3) modifier keys.

Chapter 2

Task Modeling

2.1 Introduction

In the previous chapter we showed how employing a User-Centered Design can help improve our design and thus produce *a-priori* successful systems. Actually, we can find different methods and principles that were proposed to support this approach. Similarly, multiple Model-Driven approaches were introduced. Those models were used in various ways depending on the adopted design philosophy. A Model-Driven approach in general offers multiple benefits including better analysis of the *system-to-be*, improved portability through abstraction from any implementation technology, increased productivity by enabling reuse and computer-assisted tasks, simplify proven practices, identify reoccurring patterns... It is interesting to note that most of these advantages have been found to be very beneficial in other software design philosophies such as the Model-Driven Architecture, although not yet fully implemented.

In the case of HCI, diversity lies within almost every level of these new approaches. Starting by the definition of User-Centered Design to what type of models need to be employed. However, regardless of these pitfalls, task analysis is widely recognized as one fundamental way and not only to ensure some User-Centered Design [Hackos and Redish 1998]. Task Analysis helps us to create Task Models which allow us in turn to describe users interaction with the system in a more structured way and in an exploitable form. They capture the necessary details about required actions needed to perform a task. The model further more could refine these information by defining different possible relationships that relate tasks to actions. Additionally, more information and extensions can be used according to the employed Task Model and the context of use.

In this chapter we will try to give more details about Task Modeling. In particular, we will try to answer the following questions: What is Task Modeling? How to perform Task Modeling? As for the question Why create Task Models? We chose to devote a larger space to the last question in 2.4 on page 23 to demonstrate the versatility of Task Models and their fundamental role in system design.

2.2 What is Task Modeling?

2.2.1 Origins of Task Modeling

We can trace Task Analysis foundation or roots to the "Scientific Management" movement which was concerned at the time by analyzing physical work to find a better economical design for work places and methods of production [Gilbreth and Kent 1911]. During the second-half of the last century, human and work tasks have been shifting from an industrial physical oriented approach to an Information-Oriented one. It is clear as we are approaching what most call the Information Society, *information processing* will be present at the heart of almost every day activity and task. Unfortunately, as easy and seamless this transition might appear, it brings with it additional new challenges and complications. Eventually, with this transition, human operators were and are continuing to have an increasing number of roles with varied level of complications: controller, planner, diagnostician and problem solver in complex systems [Annett and Stanton 2000]. However, such systems are prone to catastrophic failure, sometimes attributed to human error, and that is why new concepts of the limits of human performance and methods of analysis were developed [Chapanis 1959; CRAIK 1947, 1948].

2.2.2 What is Task Modeling?

Task models describe how activities can be performed to reach the users' goals when interacting with the application. They should incorporate the requirements foreseen by all those who should be taken into consideration when designing an interactive application (designers, software developers, application domain experts, end users, and managers). They are the central point where the various perspectives to be considered in designing interactive applications are combined.

Wide agreement on the importance of task models has been achieved because they capture what are the possible intentions of users and describe logically the activities that should be performed to reach their goals. These models allow designers to develop an integrated description of both functional and interactive aspects thus improving traditional software engineering approaches which focused on functional aspects.

2.2.3 Task Analysis and Task Modeling

Task Analysis is an approach that covers a set of methods and techniques used mainly by economists, designers, operators and assessors in order to describe and sometime evaluate human-human and human-machine interactions. We can define TA as the study of actions and cognitive processes performed by operators to achieve a certain goal. Thus, the primary target of TA is identifying what are the relevant tasks. It is basically an analysis activity which needs different data collection techniques, than sorting and evaluation. Different techniques can be used to collect data which can help us later analyze and identify relevant tasks:

- Interviews.
- Questionnaires.

- Video recordings.
- Observe users while doing their work.
- Check existing training and documentation materials.

While this list contains various techniques, selecting which of them to put in use is not simple. Actually there is some kind of dilemma here between high fidelity and cost. Techniques such as interviews, questionnaires have usually less fidelity than observing users. Video recording on the other hand has a higher fidelity than observing users (users usually do not behave naturally with the presence of an observer). As we employ techniques with higher fidelity, putting them into practice becomes more difficult usually due to their cost and/or ethical issues.

By the time analysts have already chosen their techniques and started collecting data, they need to identify the tasks, their number (to uniquely reference them), who will do them, and whether similar tasks are to be done more than once and by different people, for example. The final result is an informal list of tasks along their goals and additional details on how to perform them (the granularity depends largely on the goal of TA). To put this list into use, we need to create out of it more structured representations and abstractions. Those abstractions can serve us to understand better how the system as a whole works (using mainly scenarios), or they can allow us to capture some advanced details such as the relationships between tasks, goals, roles...

Task Modeling is nothing but a special type of a structured abstraction that we can apply to the results of Task Analysis. It will allow the analyst to build some *formal* models capable of capturing tasks structure, details and relationships. This clearly shows the difference between the Task Analysis activity which is concerned more about data collection and Task Modeling which serves as a way to formalize our findings. It provides a way to model tasks collected during Task Analysis into a formal model enabling more systematic uses.

2.3 Common Task Modeling Approaches

2.3.1 Hierarchical Task Analysis

Hierarchical Task Analysis (HTA) is a task analysis and modeling approach that traces its roots back to the late sixties [Annett and Duncan 1967] with the aim to evaluate an organization's training needs. This foundation of this approach is task decomposition. Tasks are considered to be logically structured in different hierarchical levels. This is achieved by breaking tasks into subtasks and actions. Since its inception, this founding idea has proven to be successful as most early as late task models have based their approaches on it. HTA was later introduced into HCI because it provides a systematic model that describes task execution making it ideal to model user's interaction with the systems to accomplish a specified goal. However, early HTA describes how tasks are related to each other in a rather primitive way. This is especially true for its Task Model definition and the employed notation (boxes for task names with numbers to indicate order and plans to describe the execution). Actually, HTA in its core is focused only on the system and its properties [Shepherd 2001] making it more system-centric. This property of HTA is a logical result given its origins and close ties with systems engineering and ergonomics.

Despite its weaknesses, HTA is considered the founding approach of various Task Models to follow. In fact, major principles such as decomposition is present in almost if not all task models today. When it comes to HTA plans, they were replaced by more *sophisticated* representation which are easier to write and to represent. More details about hierarchy and decomposition in Task Models will be provided in section 3.2 on page 33 where we will take a look at three relatively new hierarchical task models.

2.3.2 Cognitive Task Analysis and Modeling

The other major approach in Task Modeling was oriented towards cognitive techniques. It relies on findings from cognitive sciences and applies them to HCI through task modeling. This idea was motivated by the state of *"natural mappings"* between cognition and interface [Norman 2002]. Among these models we can cite the Model Human Processors (MHP) which defines three interacting systems for humans: perceptual, cognitive and motor[Card et al. 1983].

To apply the MHP model to Task Analysis, Card et al. developed a Task Model for human performance: GOMS which stands for Goals, Operators, Methods, and Selection rules. GOMS defines a set of Goals, a set of Operators, a set of Methods that are used by users to achieve their goals, and a set of Selection rules for choosing the right method among when various competing methods are available. Operators are modeled as a group of elementary perceptual, cognitive and motor actions which need to be carried in order to change any aspect of the user's mental state or its surrounding environment. The method on the other hand gives the description of the procedure needed to accomplish a certain goal. Selection rules are meant to determine which method to choose among various ones depending on the current task environment, this feature can allow us to predict which method a user will employ when confronted with a similar environment. GOMS can be used also to evaluate the quality of existing systems [Preece et al. 1994].

GOMS models produce a description of a task, often in the form of a hierarchical decomposition similar to that of HTA. However, while HTA generally describes tasks at a high-level, GOMS typically works at the *keystroke* level. We need this level of details because lowest-level operators are required to have a rigorous estimates of execution time. Thus analysts can assess system performance without extensive user testing, lowering both the time and cost required to develop a system.

Various Task Models were derived from GOMS basically to enrich it with additional features such as task parallelism and task-errors. Among these models we can cite NGOMSL [Kieras 1994] which is represented using a formal-language adding additional information to the model such as quantitative estimates of learning, CPM-GOMS (CognitivePerceptualMotor GOMS) which essentially enrich GOMS by supporting parallel execution of operators.

Except when we have already established empirical estimates for interaction (for example Keystroke Level Modeling), analysts who employ GOMS need to have a deep understanding of the foundations of GOMS mainly knowledge rooted into cognitive sciences. This makes it difficult to evaluate and create estimates that can be used later inside GOMS-based models.

2.4 Purpose of Task Modeling

2.4.1 Introduction

When we discuss what is the purpose of Task Modeling we will find a wide diverse objectives. These purposes differ according to the discipline and thus its background and goals. Among these disciplines that have special interest in Task Modeling we can cite [Limbourg and Vanderdonckt 2003]:

Scientific Management As mentioned in 2.2.1 on page 20, the *Scientific Management* movement was the first discipline to introduce the concept of Task Analysis and Modeling. The purpose was to have a deeper understanding of how tasks were performed and introduce various enhancements to improve the work and identify roles.

Cognitive psychology It focuses more on how the users use and interact with the system. Instead of focusing on all tasks, cognitive psychology puts tasks involving the user and the system ahead (Interactive Task). The analysis can help cognitive scientists to identify involved cognitive processes to perform a specific task, or evaluate needed cognitive work.

Software Engineering Task models can capture relevant task information in a formal description which allows automated processing and generation. These models can be used statically (help develop the final system) or dynamically such as to enable adaptation to variations in the context of use for the modeled tasks (Lewis & Rieman, 1994; Smith & O'Neill, 1996).

Ethnography Task models are used mainly to capture and analyze how humans communicate and interact with the system or other users probably in a specific context of use.

While Task Models are employed by different disciplines as shown above, we can find some common goals shared among them, mainly[Bomsdorf and Szwillus 1999, 1998]:

- *Inform* designers about potential usability problems.
- *Evaluate* human performance to carry a task.
- *Support* system design by providing a structured description of tasks and their relationships to other system constituents such as users and objects.
- *Generate* artifacts that can accelerate development and increase automation. For example documentation, primitive user interface (although this kind of use starts to fade) ...

Within the context of systems that are meant to be used by humans, the long-term objective of Task Modeling is definitely improving working condition (i.e. the interaction with the system) by taking into consideration Human Factors. Nevertheless, we need to be more specific on what are the short-term goals (a.k.a. sub-goals) or artifacts that we can produce out of a Task Model which in turns are going to help us achieve our long-term goal. We think that keeping the purpose clear is very important for any solution. Consequently, we will devote a larger part to discuss the purpose of Task Modeling in this chapter.

In fact, it is rare to find in the literature today researches or publications which discuss in details and in a concise manner the purpose of Task Modeling contributing more to the widespread ambiguity of implementing and adapting it in real-life situations [Jonassen et al. 1999; Dittmar et al. 2005]. By this statement, we are not disputing or underestimating the power of Task Modeling but highlighting how "confusion [still] reigns" and how the concept is not mature enough among specialists which is an old problem in TA for HCI [Anderson et al. 1990]. We can compare the situation to that of UML where every organization has its own way of using different diagrams from fully-formal use to simply a mere tool of communication. We should be clear that we do not mean multiple-purposes uses but divergent uses. To make our point clear we can talk about the high expectations that accompanied Task Modeling for a long time such as its complete generative power. It is very clear that a model has to be a generative somehow which is true for Task Modeling. Many early researches claimed that it has a complete generative power that could allow us to generate a complete interactive system out of a Task Model. Later this optimism was reduced to generating User Interfaces. These attitudes started to receive some resistance lately and new studies claim that using only Task Modeling is not sufficient for interactive systems [Navarre et al. 2009]. In particular Task Models need to be complemented with other models (mainly system ones). We will demonstrate how our Task Model can be used by other models, mainly to complement the PetShop CASE tool for system modeling of interactive systems (in section 7.3 on page 115).

Anderson et al.were critical of the situation in 1990, today, the scientific community agrees on a set of standard or basic purposes that any Task Model should fulfill, but we keep discovering everyday new ways of exploiting and using this knowledge accumulated into task models in various ways. Most of these uses are always related directly or indirectly with achieving our major goal mentioned above. In order to avoid much confusion as possible, we wanted to discuss the possible uses of Task Modeling in details. Before proceeding to detailing the set of goals we collected mainly from *scarce* literature sources and mainly industry practices, we want to underline that this is not a binding neither meant to be complete, we can always find a new way to make use of an existing model and that is one of the most powerful features of Task Modeling.

2.4.2 Discover, define tasks and remove ambiguities

The first goal of Task Modeling is identifying tasks that evolve around the system. It resembles to a greater degree to requirements elicitation where the analyst tries to discover and define the tasks of various users (stakeholders). What task modeling adds to the equation is taking into consideration the human factors and the environment in which the system will be used implying a User-Centered Design. From the last statement we can conclude that Task Modeling is not concerned by the internals of the system, it focuses on how users achieve their tasks. Thus, tasks performed by other parties, mainly the system, will be seen as black boxes and are usually analyzed in details using system models.

Along the task modeling process, the analyst will start from a high-level of abstraction down to a fine-grained description of each task, using for instance a hierarchical analysis. The extent to which the analyst should continue detailing tasks depends on various factors but in most cases ideally it needs at least a level in which all possible ambiguities are removed [Paternò 2000]. Actually the depth of analysis is may be the most important factor that separates different uses and purposes of Task Modeling. We consider this as the axis of model analysis, starting from a descriptive start to a predictive end (see 1.4.2 on page 17).

In Software Engineering, the analysis process of any methodology was always very vulnerable to various weaknesses mainly the misunderstanding between stakeholders caused mainly by ambiguities. Task Modeling help in removing them by following a well-defined analysis to identify clearly:

- What is the goal of the user (the main goal which can have itself a set of sub-goals).
- Which role (i.e. user) performs which task.
- The logical activities flow that should support users in reaching their goals.

2.4.3 Process and check most if not all cases

The strength of Task Modeling is its ability to check most if not all cases of the system use. It provides us with the required data to process and test tasks in a completely virtual environment; when referring to tools we call this operation *Task Simulation*. It is like a meta-scenario description that allows us to check all possible scenarios and alternatives that the task can follow. Thus, TM can be used to check all possible alternate paths. It is important to note that the checking is not concerned with the possible routes for a system to perform an action. The last issue is more system-oriented and depends closely on the system-design and employed algorithms. What TM allows us to check is all possible routes that could be taken *by the user*. This is possible thanks to the expressiveness power of TM to identify what are all the possible cases.

This property of TM is very valuable especially for critical systems. In these systems covering all possible cases is a requirement but at the same time represents a big challenge especially following traditional approaches. As we stated in previous sections, technology continues to evolve and new methods/-tools (tests and validation) are put in place to check all possible cases when executing a program. On the other hand, methods or tools to check the other important failure-factor human-error were scarce until the introduction of TM which promises to resolve this important feature.

When performing a task, the user can follow two different types of routes:

Normal route Following this case the user is performing according to the designer expectations. Sometimes this situation is called "best-case" route where no errors will be raised (error-free scenario).

Exceptional route The user gets into this type of routes when an exceptional event occurs (either from the user or an external factor).

We will discuss Task Flows in more details when presenting our own Task Model in 4.2.3 on page 62. For now, we will demonstrate how TM helps us identify and hence check these cases.

For the first type, TM can help us check the possible multiple normal routes that a user can follow to achieve his or her goal. To be more practical, users usually have their own method of performing the same task in the same system (for example adding a *Table of Content* in an office suite). Sometimes identifying and checking these alternate routes is very important. When these routes were not explicitly mentioned by the designer and after checking and identifying these cases, the designer can respond by:

- avoiding routes with side-effects (easy but can impact usability),
- allowing them but offer advice or redirect if possible, or
- verifying alternate routes and analyzing their criticality with the possible-errors that came along new routes (usually the most expensive but best solution).

The second type of routes usually is the most difficult to deal with. It is true that identifying and checking all possible exceptions is virtually impossible as both users and systems keep surprising their designers and users all the time. What TM can offer is allowing the task analyst to check all possible exceptional flows that could take place. As we will see in the second part, the success depends on providing well-defined flows and task conditions.

This purpose is well suited for most cases but it shows a particular high-advantage for "innovative" systems [Ozkan et al. 1998]. In this case, Task Models are used in a predictive way. They enable us to *run* the system from the user's perspective without any real implementation.

2.4.4 Cover most or all users/roles in our system

In the previous section we were concerned by all the possible cases that a task can go through. Task Modeling is a process which aims mainly to detail a task and at the same time associate it (and its description) to a specific role or user. Analysts will work on defining all the relevant tasks related to the system. During this process, they associate these tasks with a set of roles. This activity could be a progressive process where we keep discovering and identifying our users by defining their activities in our system . The Task Modeling goes further and allows us to catch more complex information mainly different relationships that can exist between those users by extending our model to Collaborative Task Modeling.

Task models are used also to assess task workload, plan and allocate tasks to users in a particular organization, and to provide indicators to redesign work allocation to fit time, space, and other available resources [Kirwan and Ainsworth 1992].

At first this relationship between Tasks and Roles seems to be simplistic but it is a very powerful feature of TM. To show the utility of it we can look how we will be able to analyze roles through their tasks. This relationship enables us to query the model in different ways in order to locate and further analyze a role and its tasks and goals. For instance, we can evaluate the role performance by defining different scopes. An example could be looking at the tasks performed by a role in a subsystem and later in the whole system. By having this flexibility we can analyze the role tasks under different situations and detect conflicts and/or anomalies. This feature can be very useful if supported with a Query Language. In Hamsters development plan, we wanted to define and implement a fully featured query language named Hamsters Task Query Language but due to time constraints our implementation remains very primitive and it requires further testing and reviewing.

In summary, this purpose is very well suited for planning and allocating tasks. Organizations can use Task Models to allocate tasks to users with a particular role. More precisely, TMs provide us with the needed indicators that can help us reallocate tasks to (1) redefine existing roles (2) identify new roles (3) avoid clashes and (4) fit to time, working place and available resources [Kirwan and Ainsworth 1992].

2.4.5 Help design the system

2.4.5.1 Evaluate the design of the system

We discussed in section 1.2 how existing design approaches tend to be technology-centered and ignore the actual user of the final system. Task Modeling can help us avoid such pitfalls by enabling the evaluation and design of a system by integrating the human element into the system design and operations more effectively. System design must consider the human as a constituent element of the system to ensure efficient and safe operation. The entire system in this case is thought of as being comprised of the following components: human operator, equipment (hardware and software), and environment.

Specifying a task model documents the order of and the logic behind the planning and organization of the tasks to be performed. This is useful for analyzing an existing the socio-technical aspects that can affect the system. In addition to determining the socio-technical aspects, the task model can be helpful in evaluating our design by allowing us to detect potential problems created by, for example, inadequate task order, disproportionate distribution of workloads between actors, or lack of time in critical phases of task execution (see 2.4.5.3 for more details specific to safety and critical-systems).

2.4.5.2 Help design interfaces and equipments

Task Models were used for a long time to support the design of User Interfaces and equipments (including input and output devices). There are even some studies which tried to develop tools that can generate a full UI out of a Task Model, tough most of these trials had several limitations and started to be reconsidered lately. It is true that Task Models are not able per se to design a complete UI, but they can be of a great help to UI and equipments designer.

The first advantage that Task Modeling offers is that the activity of UI design is now inherently Task-Oriented. The designer will build the interface based on the actions and especially flows that are defined in the model. For instance, developing a friendly assistant (popularized under the name *UI Wizard*) can be a snap with the assistance of information provided by TM.

In addition to this Task-Oriented UI design, Task Models by nature identify and show clearly the different stages of interaction that the user carries when using the system. Through a deeper understanding of the user reactions (mostly to system tasks or system output), the UI can better design his or her interface to be adapted to that particular situation. We can see the importance of this by analyzing the Classical UI design where we were limited to applying some usability guidelines. Two major things were lacking: (1) going beyond interface to interaction, (2) taking the context of use into consideration. Task Models simplify the first by making User Tasks first-class citizens and more precisely highlighting all possible interactions. Moreover, tasks are always presented in the context of a role, a goal, and a wider task. All these three help the designer put the task into context in an easy and straight-forward way.

Another interesting feature of Task Modeling is their ability to help the designer to determine accurately which elements are frequently and infrequently used and to what degree. The idea is to approach the user Mental Model as much as possible [Carroll and Olson 1987]. Having these elements in hand is considered the first key-solution to reduce clutter and cognitive overhead; taking Human Factors into consideration. This is done by hiding less frequently used elements behind some avenue of accessing those elements (like a keyboard shortcut). This process makes the UI more adapted and closer to the user's Mental Model. Actually, TM has its power in expressing things from a Task perspective but in a User-Centered fashion. Two aspects of TM make this possible: (1) TM is interested in User Tasks and include other tasks usually as black boxes, (2) TM defines task flows and especially the information flow. We will detail only point (2) as point (1) was discussed in different places above. Mental Models are usually related to how we model information in our mind. Mapping this representation to UI can reduce significantly complexity. A bad design will map directly the system variables to the UI, the reason is of course relying on system models as a source of inspiration. However, in task modeling, information is usually modeled from the user's perspective creating interfaces better aligned with the user's mental model. The flows on the other hand help the designer create interactions that are better aligned with the user's Process Mental Model (i.e. how he models different logical steps to achieve a goal).

2.4.5.3 Assess users and system safety

In critical systems, the safety of users and the system are considered the most important goals. Unfortunately, producing a safe system is also considered to be the most challenging task for the system designers. We can identify two major causes behind this difficulty:

Designing safe system The first challenge is about the method and the process we should follow in developing critical systems. This is mainly a development challenge. In this step, designers need to analyze the system and identify the risks and take them into consideration. We can say, most of the difficulties here are related to the system design. In other word it is more about technology and system functioning. Nevertheless, in designing all artifacts that are related to the user, a special attention should be given to eliminate design-induced errors.

Assessing safety The second challenge is assessing or determining how safe is the system. We cannot risk lives or major losses to carry this activity. Thus, we need a way that can help us test and validate the safety of the system.

Task Modeling cannot answer all of these needs but its contribution can be of a major importance. For the first challenge, *designing the system,* TM allows us to carry a systematic analysis of the tasks required by the user resulting in equipment that is safer to use, easier to maintain, and operated using effective procedures. Task Modeling makes it possible to capture our tasks with a flexible granularity. The details we can append to the task description can be of a various types allowing us to attach additional information related to safety and criticality in the case of critical systems. Later, using a formal analysis which takes into consideration: the user (role or agent), system and situation (context of use). In particular each element provide the system with its factors that can impact safety:

1. User: cognitive overhead, motor skills, multiple tasks, interruption, task criticality...

2. System: input, output, responsiveness...

3. Situation: depends on the application domain; generally, it takes the context of use into account.

We showed above how task models could help design better critical systems by taking into account factors that are related mainly to the user. When it comes to the second challenge, TM provides even better services to the designers. Actually this mainly where Task Models came to use in these kinds of systems. They allow us to describe the user tasks and then check them against our design and/or use them to contribute to the design in an automated fashion. This is done mainly by using simulation which enables us to assess the system in an environment very close to the real-life context without risking lives or very expensive equipments.

2.4.6 Design training programs

Today, learning and training are becoming a standard when it comes to using software systems, especially interactive and critical ones. As any complex (complex here does not mean hard to use) system, the user need to learn and spend some training in order to use the system more efficiently. In Software Engineering, documentation and training were promoted during the last years and considered among the major factors behind software success. However, in traditional approaches, documentation is usually a fully-manual activity (with the exception of reference API documentation which can be automated; here we are targeting interactive applications). This manual aspect can lead us to produce a non-compatible documentation, sometimes the change in the label of a button can confuse the user. Thus having an automated tool that can help us not only generate documentations and manual of use but help design training programs for our software are undoubtedly of a great benefit to software engineering.

Completing a task analysis can be seen in parallel as the process of identifying everything the learner will be able to do once they have completed the training. In other words, it's identifying all the content that will be included in the training in a well structured and formal way. The formal-structure help build the initial structure of the training program or documentation while semi-formal and informal details in tasks can be used to supply additional customized information [Jonassen et al. 1999] (this includes but not limited to all the name and description fields we fill inside task models). All of this is possible thanks to how our models represent the different tasks that could be performed on the system from the user's point of view.

2.4.7 Summary

We will use goal modeling which has been seen in the *Requirement Engineering* course to model task modeling purposes. The figure 2.1 on the next page represents a goal-model based on the Tropos notation [Giunchiglia et al. 2002] summarizing our findings. The goals were divided along three actors:

Designer The person responsible for the overall design of the system.

Engineer Persons who have a well defined role and required to produce artifacts for the project.

Safety engineer The person responsible for assuring that the final system would be safe to use.

Each one is represented with his/her own goals that he could achieve using Task Modeling. The most noticeable thing here is the presence of Task Models within different actors and processes. Actually this demonstrates the impact of Task Analysis and Modeling at all stages of the software development.

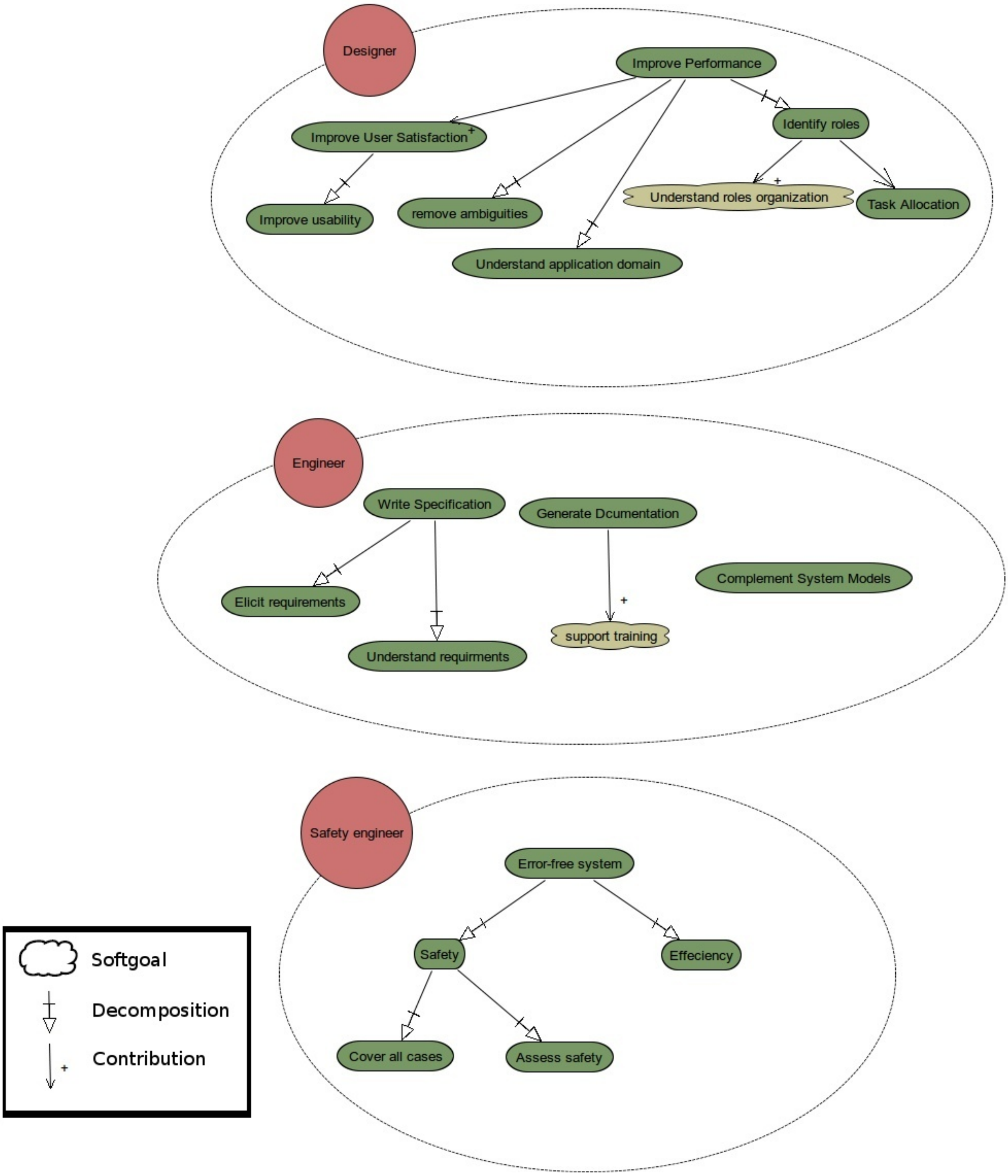


Figure 2.1: Goals Model of Task Analysis and Modeling

Chapter 3

Analysis and Classification of Task Models

3.1 Introduction

In this chapter we will have a closer look at existing Task Models. We will start first by analyzing these tasks by extracting their meta-models to understand their foundation. Then we will present their respective notation and tool if applicable. Later we will attempt to provide taxonomy for Task Modeling by attempting to provide a classification for Task Models using Feature Diagrams.

In the next section, we will limit our analysis to three major existing task models, nevertheless we will try to point common features found in other omitted TMs when appropriate. Our selection criteria was based on the popularity of the Task Model and its development continuity (a.k.a. recent task models). In the classification section, we will follow an abstract approach at analyzing Task Models features. The results will be generic so they can be applied to most Task Models.

3.2 Analysis of Task Models

3.2.1 KMAD

3.2.1.1 Presentation

K-MAD (Kernel of Model for Activity Description), known in French also as N-MDA (*Noyau du Modèle de Description de l'Activité*) is Task Model developed by Lucquiaud [2005]. It traces its roots to the MAD (*Méthode Analytique de Description*) Task Model which was developed by Scapin and Pierret-Golbreich [1989] regrouping different disciplines such as ergonomics, computer science and Artificial Intelligence. Actually, this method has continued to evolve inside the same research laboratory and some later branches of it were created until it reached the current edition proposed by Lucquiaud as K-MAD.

3.2.1.2 Model

Figure 3.1 on the facing page shows K-MAD meta-model which we extracted from the literature and KMADe (tool support for K-MAD). K-MAD is hierarchical Task Model structuring user actions and activities in the form of a tree starting from the root task to the basic elementary tasks. A *Task* has a number (machine generated ID), a name, a duration, a priority (very, rather, not very) and a frequency (high, medium, low). All these attributes are without doubt beneficial to the task description but in real-life situations some of them are difficult if not impossible to elicit. For instance, specifying the duration and/or the frequency of a future system: predictive modeling; while using them for existing system is far easier: descriptive. When it comes to roles, the task is performed by an *Executant* which can be either system, user, interactive or abstract. When the Executant is of type user, the task is assigned to an explicit agent called *Actor* who is characterized by a name, an experience level and a set of skills. Again in non-existing systems determining the experience level is not obvious. The structure of the *set of skills* is not well defined in the model and takes the form of an informal attribute which can be anything, pushing us to question how it would be used beside communication purposes.

The most important or particular aspect of K-MAD, especially compared to other models, is its strong emphasis on objects. The reason behind this choice seems to be the failure of previous Task Models to capture formally objects in their models [Baron et al. 2006; Lucquiaud 2005]. While, it is true that most models started to require formal descriptions of tasks, they are still lagging behind in describing objects and rely almost exclusively on non-formal descriptions. This lack of formality makes it difficult to link Task executions with Object states. K-MAD is trying to solve this problem by making Objects first-class citizens like Tasks. This can be seen through the different object types defined by K-MAD: Abstract, Concrete, Group, Events, Users... According to Lucquiaud, the K-MAD aims to be as formal as possible in order to simplify automated processing on the created task models. In particular, he underlines the importance and power of observing object's states change while simulating the execution of a task. Without a formal description, we will not be able to decide when and how a task influences an object.

When it comes to communicative relationships, K-MAD defines some kind of scheduler. The *Schedule* is usually attached to a parent task and specify how the child-tasks are executed. It has some attributes mainly the scheduling type: Enabling, Choice, Concurrent, NoOrder and Elementary. It uses pre-conditions to restrain task execution and iterations. When it comes to post-conditions, they are seen more like side-effects or consequences of the task execution. This definition of pre- and post-conditions in K-MAD shifts from their main purpose or *raison d'être* which is mainly testing and validation.

3.2.1.3 Notation

K-MAD uses a hierarchical representation for its notation. Each task is modeled as a constituent node in the tree with a name (with a superscript task number), an icon identifying its executant and a rectangle containing a text specifying what scheduling type this node implies on its children (see figure 3.2 on page 36 and 3.3 on page 37). When it comes to objects, they do not have an explicit

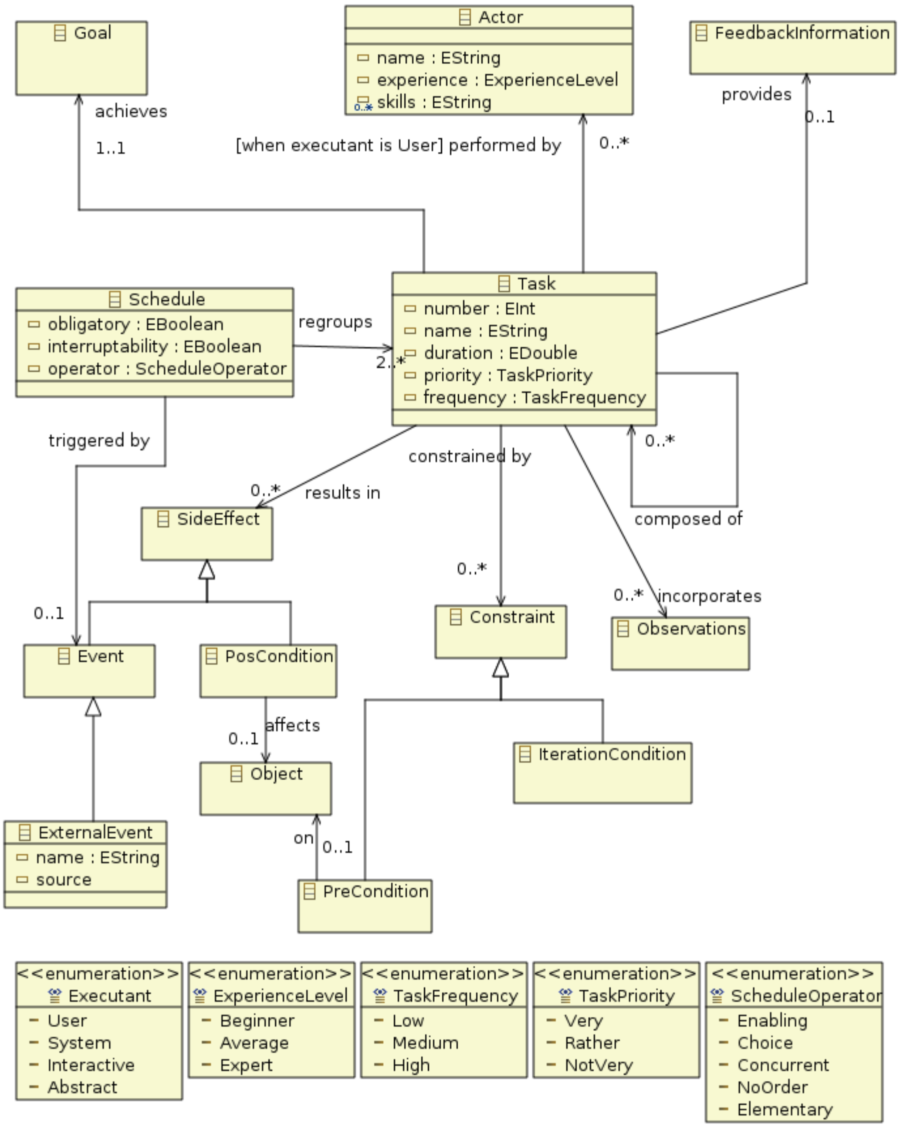


Figure 3.1: K-MAD Meta-Model

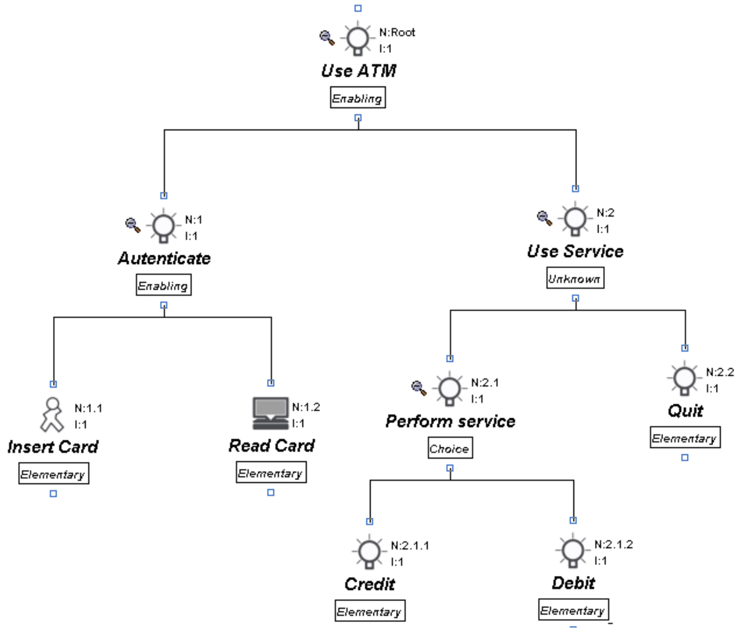


Figure 3.2: KMAD example model

notation and cannot be seen on the Task Model diagram. The tool uses tabular forms to represent objects.

K-MAD makes the task flow operator an attribute of the task itself. This attribute will affect all the children nodes. In the notation, the task scheduling is represented below the task node representation. This notation has the advantage of being explicit about the temporal operator which influences its subtasks from a central place. However, this choice represents some limitations especially when it comes to navigation. If the reader wants to know the flow type between two sibling task, he or she needs to identify the parent task and look at the scheduling attribute. Moreover, the notation employs text to describe these operators. Using text has the advantage of being unambiguous but it causes some additional cognitive work; small symbols or icons would be a better replacement.

The notation has another pitfall when it comes to modeling a complex flow between tasks. K-MAD notation allow the analyst to define one and only one operator for all subtasks. It is frequent in Task Analysis to have tasks that are of the same abstraction level but with a complex flow (two or more operators; for example a sequence than a choice). Unfortunately, K-MAD will not be able to model such cases without breaking those tasks into different levels: creating additional phantom levels and making the tree structure more complex and more importantly separating tasks that are thought to be of the same abstraction level (conceptual view) but modeled at different levels in the diagram. Another complication of such a choice is the frequent creation of superfluous tasks in order to model a complex temporal flow. These tasks are not included in the data collection repository and usually are difficult to name (as they are serving nothing but regrouping a set of tasks around an operator).

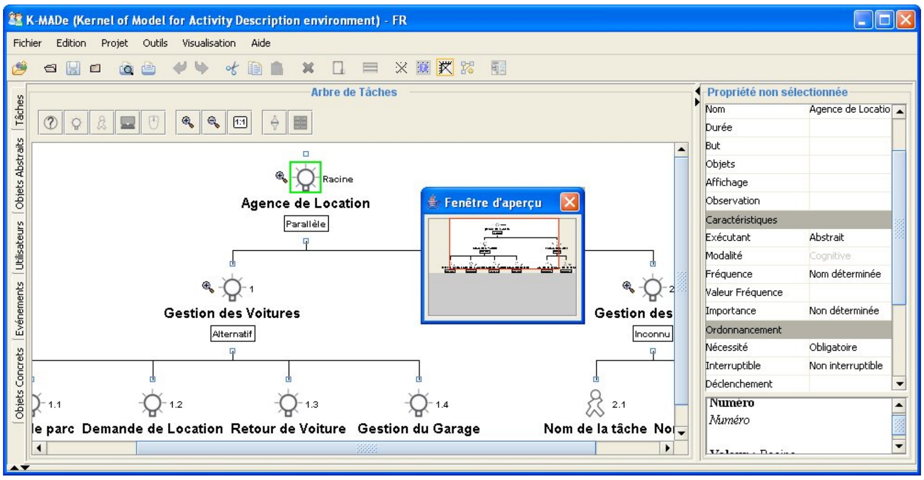


Figure 3.3: Screenshot of the K-MADe tool [Baron et al. 2006]

3.2.1.4 Tool

K-MAD has a support tool named K-MADe (Kernel of Model Activity Description Environment). Figure 3.3 shows a screenshot of the tool along the different UI features it provides. The K-MADe is noticeable for its rich interface for defining objects. Objects have their own tab along tasks. In the implementation of the tool we notice that they have their own Module, the same goes for tasks. The K-MADe tool comes with a very rich simulator too.

3.2.2 CTT

3.2.2.1 Presentation

ConcurTaskTrees or CTT for short is a Task Model created by Paternò et al. [1997] with the aim to develop a notation for task model specifications that can overcome limitations of notations previously used to design interactive applications. Thus, CTT initially was not meant to provide a new Task Model definition, or make changes to the core of previous Task Models. In its development the concern was mainly providing a better notation for task modeling. Despite this, we can identify some particularities related to this model itself.

3.2.2.2 Model

The figure 3.4 on page 41 presents the meta-model of CTT. It does not represent an official meta-model as it was extracted from existing publications related to CTT on one hand and by looking at the CTT tool which is called CTTe on the other. The CTT model and notation are known to be the most popular among task analysts and in the literature. When it comes to the model elements, CTT build on previous hierarchical models.

Tasks are usually decomposed into smaller set of subtasks. They can have a type based on to whom they were assigned: abstract (assigned to multiple agents), system, interaction (involves the user and the system), and user. The task can have different attributes: name, duration, frequency... The same criticism of K-MAD applies to CTT too about the duration and the frequency attributes for future systems. Additional attributes are the optionality of the task and recursively. An optional task is an activity that can be omitted by its agent. Although the definition is simple for this attribute, CTT does not provide enough details on how this property impact how simulation should be done. In other words, during simulation when we encounter an optional task what should the simulator do: just ask the analyst to decide or may be following a more complex path by defining other dependencies to task optionality (think of tasks that are optional but could become mandatory or system tasks that can be omitted...). CTT enables tasks to execute repeatedly but it does not define a formal way to input the type of this iteration. Thus, during simulation the analyst will do the iteration manually according to a set of rules that CTT cannot capture formally.

CTT supports objects in its model. They are usually manipulated by tasks. Analyst can define special sets of actions that they can link to objects. These actions have to belong to one of two types defined by CTT: Input Action or Output Action. These actions can be used later by tasks to manipulate objects by inputting information or outputting information. Although, CTT was keen in defining objects actions and their types, it does not provide a well formalized definition to describe objects (a pitfall that K-MAD is trying to solve).

May be the most important aspect or particularity of CTT is its temporal operators which are based on the LOTOS notation [Bolognesi and Brinksma 1987]. These operators allow the analyst to define very complex temporal relationships between a set of subtasks of the same abstraction level. Those operators are well defined and even formalized semantically using a Labeled Transition System (LTS). The table 3.1 on the next page presents the different operators defined by CTT. Most of these operators are binary, with a small set of unary ones. Binary operators means two operands, while in Task Modeling, having more than two operands is a common situation. To solve this, operators' priority was formalized, though it can be confusing sometimes. For instance if we take the choice operator ([]) by definition it is binary and it takes two operand tasks which only one could be chosen of. However, in most cases the choice could be taken out of a set of tasks (two or more). For CTT, the only solution is to include the choice operator between all the operands.

Another interesting feature that we find in CTT is its support for cooperative tasks. Usually a Task Model describes how a specific well defined role performs the task. In other words, all user-typed tasks are performed by one predefined role. However in real life, complex tasks require the collaboration of multiple roles and that is why CTT aimed to support such situations. The solution is to form the cooperative task model by creating a super-tree of existing elementary trees (role per tree). Later versions of CTT allows to include selected parts from the elementary tree into the Cooperative model (which could be simply a Task).

Finally, CTT integrates the platform type inside its model and associate it to defined tasks. For the moment, three platform types are included: Desktop, PDA and Mobile. Although, the idea at first seem to be interesting, making it an integral part of the model is not practical in all situations. The use of this property depends on the level of analysis defined by the analyst (describing

Name	Symbol	Description
Independent concurrency	$\vert\vert\vert$	Elementary actions of operands could be performed in any order.
Choice	[]	Only one task of the operands could be chosen to be executed.
Concurrency with information exchange	$\vert[]\vert$	The operand tasks could be executed concurrently but they need to synchronize their execution by exchanging information.
Order independence	$\vert=\vert$	Both tasks are executed in sequence but the order does not matter.
Disable/Deactivate	$[>$	The start of execution of the second operand task abolishes the first's.
Enable	$>>$	The end of the execution of the first operand starts the execution of the second.
Enable with information	$[]>>$	Same as *Enable* but the first operand can send information as input to the second operand.
Suspend-resume	$\vert>$	The second operand can interrupt the execution of the first. Once it is done, the first task could resume execution.
Iteration	*	Allow the task to execute repetitively. The iteration can be stopped when the task is *Disabled* ($[>$) by another task. CTT allows the analyst to give the number of repetition too when it is known (replacing the * by n; iteration count).
Recursion		Same as repetition but the context is not reset each time the task is re-executed.

Table 3.1: Temporal operators defined by CTT

a task independently of the platform). In addition, we can have tasks that can be performed from different platforms which is becoming a software trend lately. We think that this property can be of a better use at lower levels of the system design. This will contribute to the abstraction of tasks from their system implementation and platform-dependent factors. While we understand that the CTT model will be used primarily to support multi-platform system interfaces, we do not encourage the introduction of such properties as an integral part of the core meta-model.

3.2.2.3 Notation

As mentioned in the presentation, CTT is more about notation than the model itself. For its notation it uses hierarchical structure of tasks. Each task is represented in a node with an icon indicating its type (abstract, user, system, interaction) and a label indicating the task name. When it comes to objects, CTT does not provide any notation to represent them inside the diagram. Analysts can enter details related to objects only through dialogs.

What set CTT apart from other notations is how it represents operators. In addition to the parenting links (vertical from top to bottom), CTT adds other explicit links between tasks of the same level (horizontal from left to right). These links are accompanied by a symbol (see table 3.1 on the preceding page for a legend) indicating the temporal operator that relates its two operands (see figure 3.5 on page 42 for an example). The immediate advantage of such a notation choice is the ability to define complex temporal flow between a set of subtasks without breaking them into additional levels as in K-MAD for instance. Thus, CTT diagrams are usually smaller in size (reduce the need of superfluous levels) and closer to the conceptual representation of the task model (tasks of the same abstraction levels are always at the same level). However, there are some weaknesses in this notation. The most important is the order of execution. When we have a set of successive operators we need to know in which order they should be executed. To solve this problem, CTT introduced operators' priority, nevertheless from a notation perspective analysts would not always escape from confusion from time to time. Some solutions were proposed aiming at making operators priority more explicit by copying the concept of parenthesis in arithmetics.

3.2.2.4 Tool

The CTT tool is named CTTe (ConcurTaskTreesEnviroment). It is a rich Java Swing application that allows analysts to write their task models. Some of the interesting features it provides is its ability to import scenarios and extract task models from their descriptions. This feature can be useful to help the analyst have a basic version of the task model to build-on later. One thing to be careful about is the terminology incompatibility between the tool and the model description in published articles (especially newer version tend to use different terms not necessary the same employed in older publications).

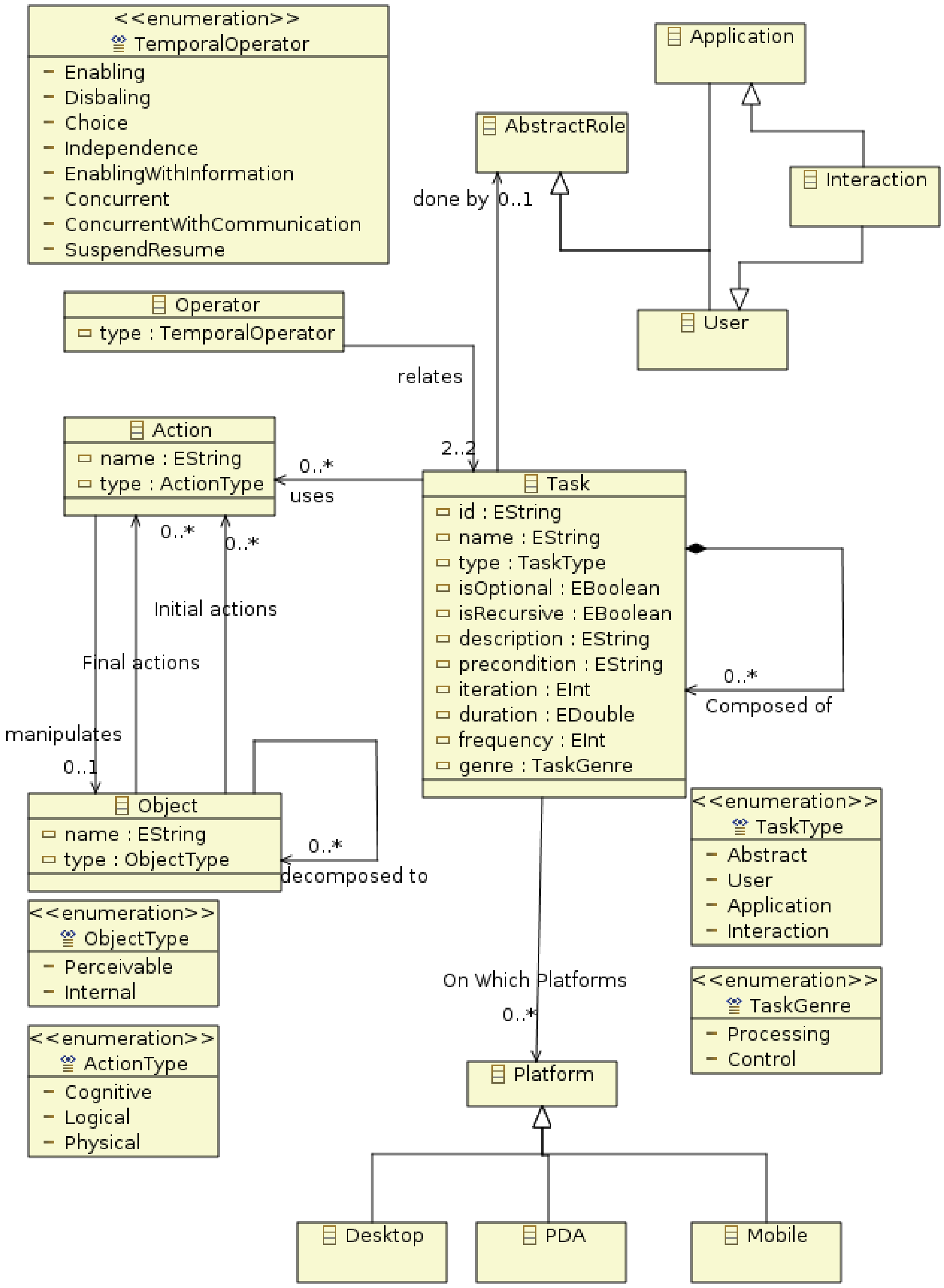


Figure 3.4: CTT Meta-Model

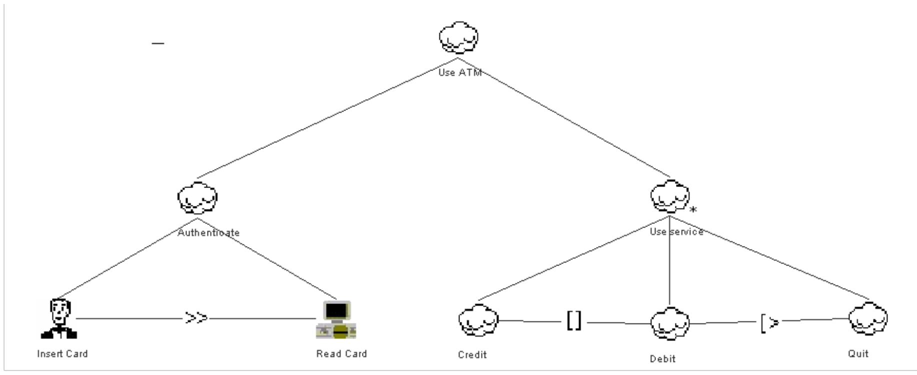


Figure 3.5: CTT example model

3.2.3 AMBOSS

3.2.3.1 Presentation

AMBOSS is a hierarchical task analysis model and tool created especially for safety critical systems by a team from the Institute for Computer Science (University Paderborn, Germany). It was developed mainly to create a Task Modeler that takes into account the particularities of critical systems mainly the safety and the socio-technical factors [Giese et al. 2008]. The model differentiates itself from the others by highlighting the additional information and "appropriate structures" it appends to Task Modeling. These additions are primarily concerned with aspects related to time, space and communication.

3.2.3.2 Model

Additional elements related to safety make the larger part of the AMBOSS meta-model which we extracted from [Giese et al. 2008] and its supporting tool named AMBOSS (see figure 3.6 on page 44). The first major concept that was added to the model is the *Barrier* element. Barriers are found to prevent harm primarily to human beings but also equipments and materials in general [Hollnagel 2004]; they can be physical such as a protection suit from dangerous rays or more abstract like laws. However, in task modeling, the focus is user tasks implying that introducing the element of Barriers can be of a limited impact in user modeling. This is due to the type of barriers that we can include in such models which are social in nature and can be breached easily by users; we cannot have a systematic check for these barriers. We think barriers are meant more to be included in system models as they will be an integral part of the logical execution and will never be left without being checked. Furthermore, barriers in task models can be modeled in an easier fashion as pre-conditions or guards to tasks.

The other major addition we can find in Amboss is the information flow. Messages are exchanged explicitly in an Amboss model and they can carry different types of information to the destination task. To make the concept safer, the principle of feedback was added allowing the sender to verify if a message has arrived or not; even get a more structured response.

As for communicative relationships, Amboss defines a set of basic temporal operators: sequence (SEQ), serial (SER; execution is arbitrary), parallel (PAR), alternative (ALT; only one subtask could be executed), and SIM (all subtasks have to start before any subtask may stop. The temporal operator is assigned at the task level and is applied to all subtasks. All the defined operators are easy to understand and employ with the exception of the SIM operator. Besides its semantics, the condition it verifies was not given enough arguments or examples to demonstrate its usefulness.

The other thing to notice in Amboss, when compared to other models, is the introduction of the spatial dimension into the model with the *Room* concept. The space dimension can be of an enormous importance for some critical systems but mostly it is not the case. We think it would be better to consider this aspect as an extension more than an integral part of the core model. Additionally, the room can be seen as a specific case of conditioning to the execution of a task. Thus, it can be expressed inside the model without explicitly having its own classifier in the meta-model.

3.2.3.3 Notation

Amboss uses a hierarchical representation for its notation. Each task is modeled as a constituent node in the tree using a rectangle with three compartments. The first compartment contains set of icons that play the role of flags; for example indicating whether the task is critical or not etc.. The second compartment contains a text reflecting the task name. The third and final compartment shows what temporal operator this task applies on its subtasks (see figure 3.7 on page 45). The same critics related to temporal operator placement in the model which we mentioned while discussing K-MAD notation applies here (see 3.2.1.3 on page 34).

Amboss exploits the horizontal links between subtasks in a different way than that of CTT (which uses them for temporal operators). They are used to show the information flow between the different tasks (regardless of the level). Messages are modeled with a small circle inside the link.

3.2.3.4 Tool

The Amboss tool is rich in interaction when compared to other tools. Particularly, it allows the user to manipulate most attributes of the element directly inside the diagram without requiring him or her to visit the property panel each time. This is mainly due to the reliance of Amboss on the Eclipse Modeling Project tools to develop their tool.

The Amboss tool provides a simulator and a graphical query-builder. The latter can be used to write queries graphically and interrogate task models.

3.3 Classification of Task Models

In this section, we propose a feature model to compare different task models approaches and provide more formalized criteria to categorize them. The feature model makes the different possible approaches in Task Models more explicit. We are not going to detail every classification feature as most of them have

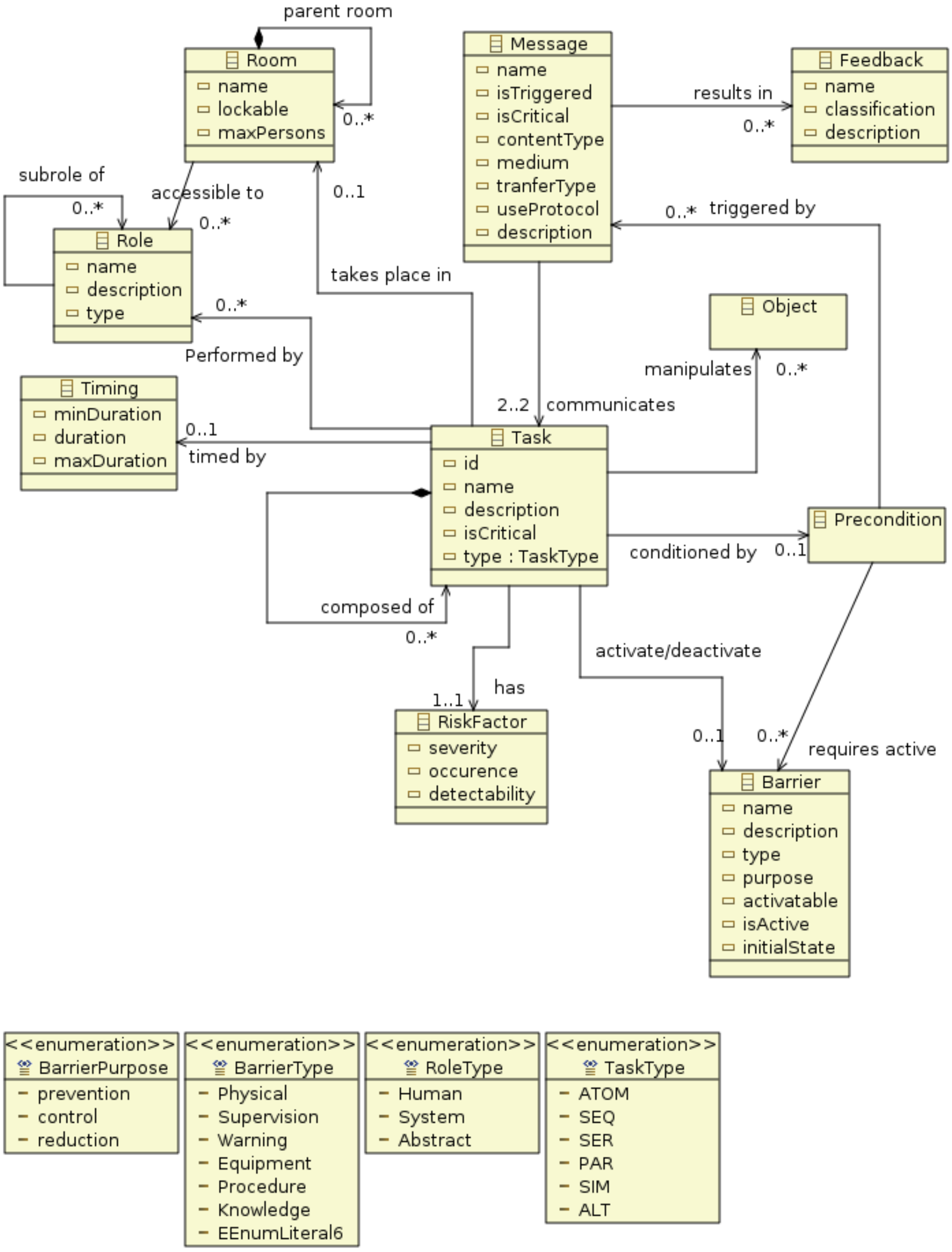


Figure 3.6: AMBOSS Meta-Model

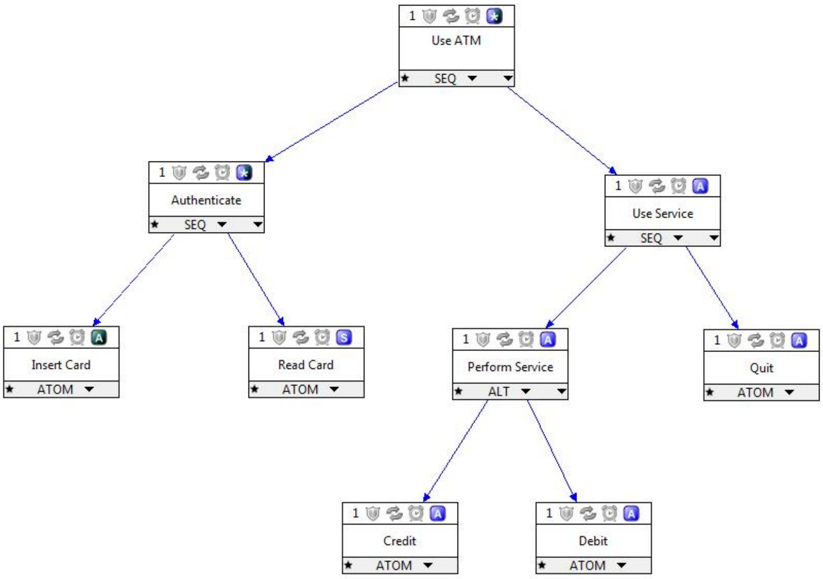


Figure 3.7: Amboss example model

been discussed or will be; references to related sections will be provided accordingly. In order to make some features clearer, we will provide concrete examples (whether from an existing model or through more general cases). Identifying all these features will allow us to form different classes for different strategies. The importance of this classification lies in its generic applicability; it could be used to classify future Task Models for instance although this does not mean it is complete. This section is organized as follows, it will start by presenting the method we used to create our taxonomy which is feature models. Next we will present the different classification axis we have collected with a stronger emphasis on features related to the model.

3.3.1 Feature Modeling

We will rely mainly on a special analysis technique called Feature Modeling [Czarnecki 1998] to provide a classification for Task Models. Essentially, a Feature Model defines taxonomy to classify a set of elements from a specific Analysis Domain. In our case, the domain is Task Analysis and the elements are Task Models. A key element of the feature model is the *feature diagram*, which is a graphical notation for describing dependencies between (variable) features.

Feature Diagrams are proven to be particularly very useful when developing new models and languages [van Deursen and Klint 2001] which is the case here. The following classification will form the basis for our model Hamsters. Mainly, alternative choices and different techniques which could affect the development of a Task Model will be synthesized here from existing Task Models features and particularities. Before proceeding to the next section, we will present a generic high level Feature Diagram for Task Modeling along a legend in figure 3.8 on the next page.

As you might notice, two major classification axes were analyzed:

Model This branch of the feature diagram will list all the features that are related to the definition of the core model. Mainly, it presents the different required features and/or alternatives we can include in our meta-model.

Notation Features related to notation are described here. It gives us an overview of the possible language choices that we can use to represent our model in a human readable format.

It is possible to include additional axis such as Simulation presenting all features related to simulation choices and how it can be carried. Finally we could also include a fourth axis which classify based on use purposes, but this will result in reproducing the same figure 2.1 on page 31 which uses instead a better adapted goal-modeling notation.

3.3.2 Model

3.3.2.1 Model Structure

The first feature is related to the structure of our model. The structure of the model defines how the different task are related from a conceptual point of view. It maps to the decomposition relationship in Task modeling. Two alternative structures are identified:

Hierarchy The most adopted structure. It was initiated by the HTA task model. The idea is to represent the task in the form of a hierarchy (tree). Usually a parent task is decomposed into children nodes which are called subtasks. It is easy to implement and more importantly considered easier to represent and process. The popularity of this choice is due to the claim that that people find hierarchies naturally easy to understand [DeMarco 1979]; Paterno, the developer of CTT, claims that people's understanding of hierarchies is "intuitive".

Heterarchy A more complex structure, which actually is a more generic form of hierarchies. In fact, a hierarchy is a special case of heterarchy. The most important feature of heterarchies is their ability to allow its nodes to be related to more than one thing. In other words, a node can have multiple parents or better described can be included in multiple nodes. The reason behind this complex structure is how our world is really organized. According to [Diaper 2000], it is far less clear that either the natural world or the social one are arranged hierarchically. Although this representation of the world seem to be more accurate, its adoption by the Task Modeling scientific community remains limited. We can cite as an example of Task Models using this structure the TAKD model (Task Analysis for Knowledge Descriptions) [Diaper 1989; Diaper and Johnson 1989].

It is clear that the second choice represents a better fidelity and flexibility to represent real-world models. In the context of tasks, heterarchical models are quite powerful in modeling tasks that are shared among various higher abstract

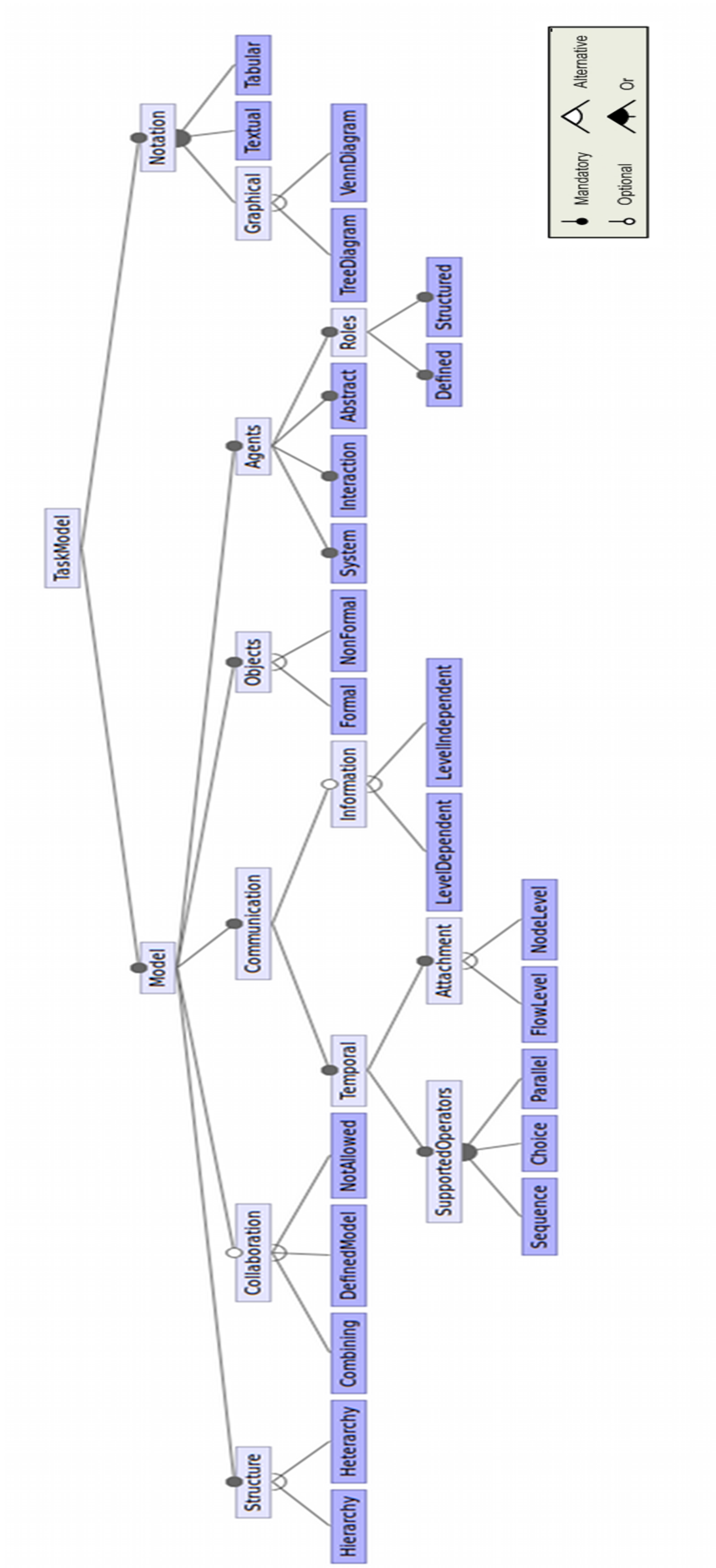


Figure 3.8: Generic Feature Diagram for Task Models

ones. Current trends seem to prefer the first structure for its simplicity and formal-imposed nature (it pushes the analyst to think more about what constitute a task). In some situations, exactly *incertae sedis*, heterarchical models are to be used instead.

3.3.2.2 Collaborative Task

The model should be able to support collaborative tasks. Those are tasks that cannot be performed without the participation of multiple roles. In our feature model, we identified three alternatives:

NotAllowed Simply, the model does not support collaborative tasks.

Combining Using this technique, the model constructs collaborative models by combining existing atomic models. The analyst will select the relevant role-based tasks required to achieve the task and then will combine them to form a new model representing the different atomic models along an additional information indicating to which agent it was assigned.

DefinedModel In this case, the model goes beyond a simple combination but add additional features specific to collaborative tasks.

3.3.2.3 Communication between tasks

The model can define a set of possible communication flows between tasks. Those links are usually called communicative relationships. In Task Models two major relationships of this type are found:

Temporal Those relationships are used to define how a set of tasks should be executed. The model can define various operators such as sequence, parallel, choice ... The model can view the operator as simply an elementary type and thus it cannot provide additional attributes related to this operator. Operators can be defined as elements too and can have their own attributes related to how they carry the execution in details. Most of the Task Models we discussed, consider the operator as a type and does not take the specificity of each operator. In the K-MAD model, the operator is called schedule which has some common attributes among all Schedulers like optionality and interruptibility. Another feature related to temporal operators is where they should be attached. A first alternative attaches the operator to the parent node (e.g. K-MAD, Amboss). A second one attaches them to a flow link (e.g. CTT).

Information Exchange The model can define a way to allow exchange of information possible between tasks. Information here can be of any type from scalar to complex objects. The model can make this feature available only to tasks of the same abstraction levels (has the advantage of tasks encapsulation) or LevelIndpendent. In the latter case, any task can communicate with other tasks from a different level in the same hierarchy or even a different one. Amboss is a typical example of *LevelIndpendent* information flow named *Message*.

3.3.2.4 Objects and Agents

The model can include objects which can be manipulated by tasks. Most models allow the capture of objects but do not provide a formal representation of objects. In particular, there is no consensus whether objects should be modeled as *classes* or *instances*. In addition, the way we should capture the attributes of an object remains most of the time informal (with the exception of some models notably K-MAD).

Agents are used as the major classification criteria for tasks in most models. Depending on which agent should perform the task, we determine its type. Four major agent types are widely defined: abstract, system, interaction, and user (role). The model can add additional details related to roles; at least a name. Some models allow the analyst to provide the organizational structure of its roles (can be useful for safety critical systems where the role define also the level of clearance); it can be for instance the organogram of an organization.

3.3.3 Notation

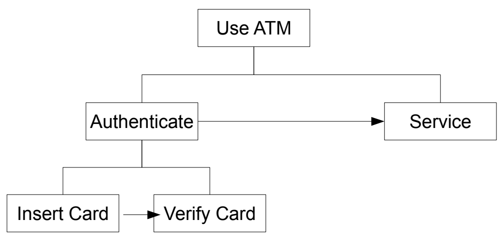
Different notation features are possible for Task Modeling. They depend heavily on the choices made at the model level. As any other model, a TM can be modeled using different languages which can be graphical, textual, tabular... In our feature diagram, we will concentrate more on the graphical notation.

To model the structure of the model graphically, we can opt for different options depending on the structure of our model. For hierarchical models, the most widely used graphical representation is a tree diagram. Another alternative is employing a Venn-based diagram which is based on the set theory.

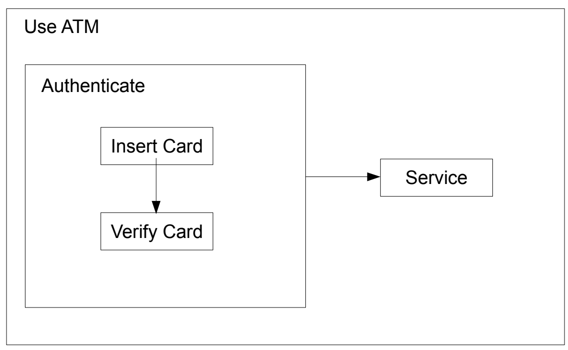
Tree diagram Tasks are structured in the format of a tree starting with higher level tasks from the top to more detailed ones in the bottom. Conceptual relationships are created using vertical links between the nodes (associating the parent to its children). Communicative relationships are usually drawn horizontally to represent a flow between subtasks.

Venn diagram This notation is based on the set theory. Instead of using nodes and links, the Venn-diagram represents elements as sets and uses the *contained-in* graphical relationship to identify the parent set (or task in our case); conceptual relationship. For communicative relationships, arrows linking different sets are used to represent flow of tasks.

The figure 3.9 on the next page represents the same model using both a tree diagram (3.9a) and a Venn-based one (3.9b) . We will not develop further features related to notation in this section as we will discuss notation in Task Models in details in chapter 6.



(a) Tree-Based Model



(b) Venn-Based Model

Figure 3.9: Model Structure Notation

Part II

Hamsters Task Model

Chapter 4

Foundation

4.1 Task Structure

4.1.1 Dealing with complexity

In this part, we will represent how our model deals with complexity. We will explain the basic methods and techniques used to make it easier to model and understand task models.

4.1.1.1 Abstraction

In modeling, abstraction is a key tool not only to simplify the modeling process but to focus on specific parts of what we are modeling. Task Modeling is not an exception and relies heavily on Abstraction to deal especially with the complexity of tasks carried in a system. Hamsters abstracts the task models in its basic form by:

Abstracting the system: When carrying task analysis, tasks related to the system are not required to be detailed. Actually, those tasks should be modeled as black boxes because the Task Model should represent the Task from the user's perspective. This abstraction allows the analyst to concentrate more on user and interactive tasks without giving much importance to system ones. In worst cases, he should know what is the awaited input or the generated output from a system task.

Limiting analysis to one role: In early stages of the task analysis, Hamsters should scope the analysis of a specific task to a one role. Thus eliminating complicated tasks involving multiple agents and complex social interactions. Focusing on one role allow the analyst to provide better details and elicit the particularities of that role for performing the required task.

Optionality: Hamsters aims for generic use. The goal is to provide a Task Model that can be used for different purposes and does not limit itself to one use. This generic feature is achieved by allowing optionality and extensions. Optionality defines a set of profiles for different types of system designs. Each profile defines the attributes that are relevant to that domain and omits the rest or makes it optional. Extensions are special structures that provide additional data to the meta-model, their purpose is complementing the model with domain-specific features not found in the core model.

4.1.1.2 Decomposition

The aim of task decomposition is to decompose the high level tasks and break them down into their constituent subtasks and actions. This conceptual relationship helps the analyst build the overall structure of a main user task. Decomposition presents our task at different levels of details or *abstraction*. At higher levels, tasks seem to be more abstract (We mean Task abstraction not the model one). At lower levels, we opt to make our structure richer by identifying less abstract tasks and defining the different flows that relate tasks, mainly from the same level (which are basically communicative relationships in nature).

The process of task decomposition is carried out by identifying subtasks. In order to break down a task, the analyst should ask a first question of type "how". The answer will help him/her formulate a description of the task. Following this, he or she should extract the steps required to achieve the task out of this description. Alternatively, we can perform the decomposition in the reverse order by building-up the tasks. In this case the analyst starts with the basic tasks and identify higher tasks by following a goal-oriented analysis. In other words, the driving question of the analysis would be "Why this task is needed?". The extent to which the analyst should decompose a task depends on many factors. More detail will be provided in section 4.1.3 on page 56 which discusses abstraction levels.

4.1.1.3 Projection

Task Models can vary in size considerably, but most of them are employed to model interactive-systems with a higher interest that can be seen from critical ones such as Air Traffic Control. Those systems tend to be very complex and would require a well sized model in order to capture all the tasks related to them. To deal with this kind of complexity, Hamsters aim to allow projections on task models. A projection aims to select a sub-part of the whole model and analyze it in isolation of the rest of the system.

In Hamsters *projection* is carried out using two methods:

Simple projection Those are simple projections that identify smaller parts of the system based on simple criteria or model partitioning. Simple criteria can be a specific role or a task at a predefined level. Those projections are built-in the hamsters model.

HQL Hamsters Query Language is an advanced projection tool that allows analyst to execute queries on the model to select a specific part. In addition to projection, HQL can be used to perform profound analysis and evaluation on the system. In fact, our thesis does not provide details about HQL as it was not fully implemented, it is considered as part of the prospects.

4.1.1.4 Modularization

Hamsters aims to be a modular model. We mentioned already in the *abstraction* section how flexible and generic Hamsters tend to be, mainly through Optionality Profiles and Extensions. Extensions are small model definitions that can enrich our core model to decorate it with additional formal information. Depending on the application domain, the extension can require additional details from the analyst and more importantly capture it in a formal way.

Modularity of Hamsters is not only about the possibility to extend its metamodel but also lies in its foundation. Hamsters makes it possible to develop modular models that can be used in different situations. To achieve this, two techniques are used:

Reusability: All tasks defined inside a Hamsters model can be reused in different locations. The reused task can be either plugged directly or referenced.

Patterns: Hamsters aim to capture reoccurring patterns and to simplify their introduction into future models. In ideal cases, we aim to provide some refactoring capabilities.

4.1.2 Conceptual relationships

Conceptual relationships specify how the main elements of our model (tasks) are structured. The most important conceptual relationship in task modeling is the *has-a* (or *decomposed-into*) relationship. In our classification of Task Models, we identified two major structures for task models (see 3.3.2.1 on page 46). The heterarchy which represents a closer model to how our world is structured. The hierarchical structure on the other hand tends to model our world in a more organized simplified way (regulation and simplification of reality). The choice between adopting a hierarchy or a heterarchy is not a problem that concerns only Task Modeling, almost every model of the world should ask this question when defining its conceptual relationships. If we look at the discipline of Software Engineering in general, we will find that hierarchical models prevail. The argument behind this choice is usually that these models are a *deliberate* simplification of the real world. However, such hierarchical models pose some serious problems and the reason is the high likelihood to have an invalid model structure [Diaper 2001] that does not reflect the *thing* we are aiming to model.

No doubt that from a theoretical point of view, Heterarchical models seem to be very tempting with their ability to provide better models with higher fidelity, but their problem lies in later stages of the model development: precisely how they should be represented? When it comes to notation, Hierarchical models outperforms heterarchical models in almost all levels: simple to represent, easy to implement and highly popular in different domains. Heterarchies do not have a well defined graphical representation when compared to the popular hierarchical graphical representation in the form of trees. The solution can be replications for trees; whenever a node is linked to another one, we duplicate it as necessary. However, this solution can make our diagram looks cumbersome and rise confusion as multiple nodes of the same task are represented. Heterarchies are easy to represent mathematically (using the set theory) and in computing (using the graph theory) but they tend to be difficult to represent graphically which is an important aspect for the success of any model in nowadays.

In Hamsters, we opted for a hybrid solution. We will use heterarchies in an indirect way inside the model structure but well employ hierarchies for notation. In the notation, we chose to represent nodes with multiple links using *references* (node replication but with explicit semantics for the user). References are a special type of tasks that simply call another task (like a proxy who passes the execution to the task it is referencing). Another sophisticated solution for the notation lies in the use of Venn-diagram instead of trees. The power of the Venn-model is that it is a set based model (uses mostly as a graphical notation in the Set Theory). We can look at tasks as sets and those sets can share elements enabling us to define tasks with multiple links. More details about this notation including its advantages and shortcomings can be found in the notation section.

4.1.3 Abstraction Levels

As stated above, Hamsters allows the analyst to decompose the task into a set of subtasks creating in the process different abstraction levels. Each lower level represents the task with a certain added details. Introducing abstraction levels into the model helps the analyst identify smaller parts of tasks with every new level, creating a structured task model in the process. Usually this activity depends on the data collection phase which helps us to identify how tasks are or should be carried. The most important question for abstraction levels is not how to break down a task into subtasks but when to stop decomposing tasks. The question can be reformulated to be: At which abstraction level should the analyst stop? Deciding upon the level of detail into which to decompose depends on various factors. Those factors are related mainly to (1) the type of the system to be modeled and (2) the way we intend to use task models later.

The system type impact the required level of details of the model. For instance in critical systems, having as much as possible detail is very important to understand the system and more importantly to increase the accuracy of evaluation and assessment. Primitive tasks in this case should be well defined and unambiguous. Other types are less concerned about much detail but focus more on enhancing existing systems performance. In this case, the analyst focuses more on task allocations to roles and understanding the required process to achieve a task.

The second factor that has a great impact on the level of details is related to the defined purposes for the task model. If we aim at evaluating required performance, a very deep level is needed. In this case, the model can reassemble to a GOMS model and the analyst could provide very low abstraction levels such as KLM (Keystroke-Level Model). If we intend to use the task for communication purposes only than a higher level would suffice. It all depends on how we will put our task model into use later. Here higher level models have the advantage of being modular thanks to their increased abstraction. They can be used and reused in various systems. Whereas, lower-level models are more coupled to the system design and tend to be less modular and manageable.

Unfortunately, those factors cannot be encoded inside the Hamsters model to help the analyst choose the right abstraction levels. Choosing the right abstraction is considered more an art than an exact science. Nevertheless, Hamsters can be useful to check some aspects of the chosen level of details. For instance, Hamsters can help the analyst ensure that all the subtask decompositions are treated consistently after the end of a modeling iteration. This check can be performed by looking at the flows details at the lowest level of different tasks. An initial check would verify that the output of one subtask is the expected input of another in a different but related hierarchy. But this is not easy as it might seem. Relying only on decompositions and flows is not sufficient for a complete check. To make the implementation more sophisticated, Hamsters consider levels as entities in the model having their own properties. Thus, it does not look at levels as simple conceptual relationships that are the logical consequences of creating a new subtask. Abstraction levels in Hamsters, could have additional attributes that give some formal semantics to them, which can be in turn used for various purposes later.

Note that this view of abstraction levels in Hamsters is flexible and it does not need to be explicit. This is achieved through a default behavior: the creation of a new subtask leads to the creation of a new level if the parent has no subtasks. Those subtasks will be assigned implicitly to this new level. However if we think of a scenario where we have two distinct sub-hierarchies, we will come to an interesting question: How the levels of the first hierarchy are related to the second? The simplest answer would consider that tasks having the same *decomposition level* would belong to the same *abstraction level*. Actually, this is not true in all cases. The way we detail two tasks can be different. Some tasks tend to be more complex than others. Complexity here is not about the number of subtasks but about the required number of decompositions in order to reach the required level of details. To make our point clear we will use a simplified real-life example modeled in figure 4.1 on the following page (the only aspect we are interested in this model is its levels, details about the notation of our model will be provided in 6.2 on page 89). It models how a user can use an ATM, it focuses precisely on the withdraw task which is enough to demonstrate our point. First we identify two main related hierarchies issued from the same parent task but with different complexities (or decomposition levels): *"Authenticate client"* and *"Use service"*. The first has only one additional decomposition level while the second has two (with the additional abstract task *"Withdraw money"*). If we consider that decomposition levels are identical to abstraction levels, than the subtasks of *"Authenticate client"* would have the same details level as the *"Use service"* which is not true (the latter being abstract and the others being grained). Hamsters aims to be intelligent enough to differentiate between decomposition levels and abstraction ones. However, its ability to distinguish them is limited mainly to the lowest level. For intermediary levels, the user can specify explicitly the abstraction level that a decomposition belongs too; that is why levels are seen as entities in Hamsters. Provided with these information, Hamsters can run various checks on the model to validate its consistency. The most important check is the complementarily between subtasks of the same levels coming from various hierarchies. If when combined, those tasks forms a well formed task flow model than the check passes otherwise it fails.

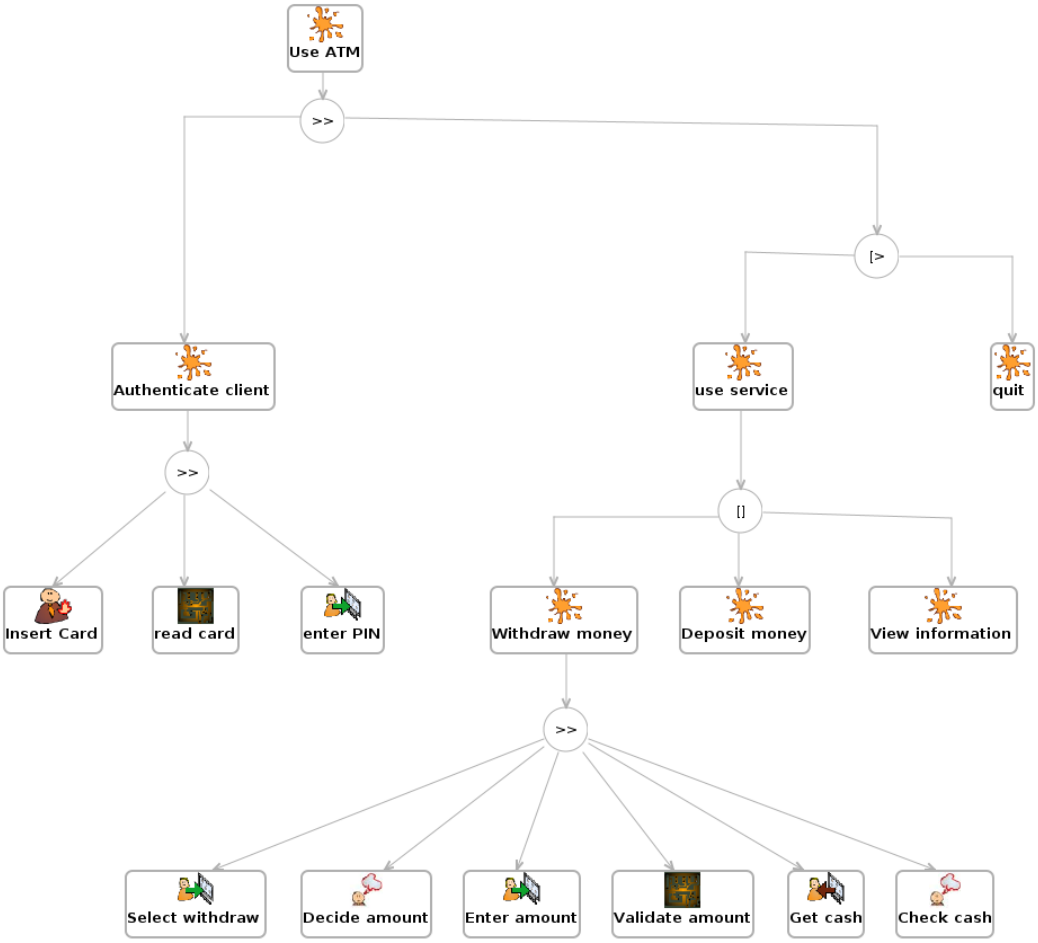


Figure 4.1: Example task model to withdraw cash from an ATM

4.2 Communicative relationships

4.2.1 Introduction

Hamsters allows analysts to create two types of communicative relationships. The first is related to how tasks are related in time (mostly temporal operators) and the second represents the information flow between tasks.

4.2.2 Modeling temporal operators

Temporal operators provide the necessary details about how tasks should be related in time. Those relationships can be simple as a sequence of actions or more complex such as precise synchronization between two or more tasks. Temporal operators in Hamsters are attached to the parent Node. This operator will define how all its subtasks are related in time. Our choice is similar to that of K-MAD or Amboss (see respectively 3.2.1.3 on page 34 and 3.2.3 on page 42). The advantage that attracted us to this choice is the explicit presence of the operator and how it helps eliminate confusion which can be noticed in CTT models with the operators' priority problem (see 3.2.2.2 on page 38). Meanwhile, we need to solve some problems inherited from our choice. These problems are concerned mainly with superfluous levels and tasks when modeling complex task flows.

In classical approaches, the analyst must create a new task each time it needs a new grouping (think of parenthesis in arithmetics) resulting in a new decomposition level and a additional cognitive overhead to find a name for the new task and relate it the conceptual view of the model (constructed from data collections). After adopting operator's attachment to the parent task, a debate arose about how to model this modification inside the meta-model. Two alternatives were possible:

4.2.2.1 Both tasks and operators should be seen as nodes

In this solution, the model will be simply a generic tree model (like DOM for instance). Our task model will become a set of nodes having the following classifiers (note that we will limit the discussion to the Task Modeling level; see 5.2 on page 66 for more details about Task Modeling Levels in Hamsters):

Node An abstract class (similar to the Node interface in DOM) used as a super-type for any element in the hierarchy.

TaskNode The node that will be the super-type of all tasks defined in the meta-model. (It plays the role of the AbstractTask in this case).

OperatorNode A special node to represent an operator.

The final result would be a tree that contains two different types of nodes: *TaskNode* and *OperatorNode*. The model will be definitely a hierarchy but *not* of Tasks. It will be a Tree of mixed elements as semantically speaking a Task is different from an Operator: a Task is not an Operator and an Operator is not a Task. Making them having the same super-type does not make any sense in this regard. Any element introduced in the model has to have its semantics well defined, what is the added value by adding the Node classifier other than its utility in our notation meta-model? The problem with the solution lies in the fact that it mixes the front-end or the notation meta-model with the domain meta-model. For the analyst, the model is not seen as a tree of Tasks and Operators but rather a tree of Tasks related according to the operators he added; for him it is always a hierarchy of tasks. To express this argument from a theoretical ground, by making an Operator element of the hierarchy, we are mixing Conceptual relationships with Communicative ones. In a conceptual relationship, the goal is classifying or categorizing elements (through generalizations and specializations), whereas communicative ones are used to define the flow of communication between different elements (the case of an operator in Task Modeling). Another major difference between the two relationships is that the first is structural in nature (defines the hierarchy of our tasks), and the second is behavioral; it defines how our tasks are meant to be executed and exchange information.

To better see the picture we will use the following sample model:

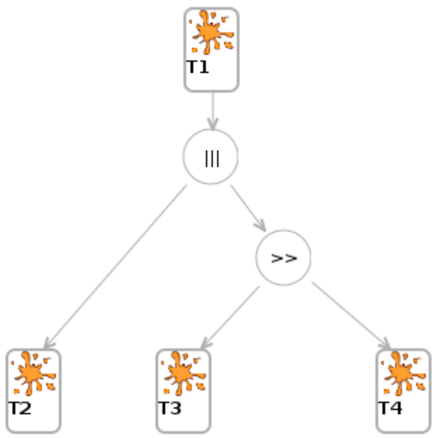


Figure 4.2: Sample model for temporal operators

We will use a pseudo-XML language in order to show how the model is stored inside the system (only information related to the model are found here, data related to the notation is added in at lower levels). Our sample model is stored in this structure following this solution:

```
<taskmodel>
    <roottask name="T1">
        <subnode:operator type="|||">
            <subnode:task name="T2"/>
            <subnode:operator type=">>">
                <subnode:task name="T3"/>
                <subnode:task name="T4"/>
            </subnode:operator>
        </subnode:operator>
    </roottask>
```

```
</taskmodel>
```

Listing 4.1: Sample model structure (*Operator* is a node)

By analyzing the above domain model representation of our graphical model we may notice:

1. The task model is nothing but a tree of semantically different nodes.
2. We will have an additional superfluous decomposition level at each task. In our example T2 and the operator ":]" are not the direct children of T1 as the user might think (even if the user does not necessarily perceive this as an additional decomposition level).
3. We will have always Tasks that have only one child no matter what we do (in our example look at the lonely child of T1 which is "|||", this operator cannot have any sibling in this case).
4. What about if we want the execution of operator ":]" to be optional or make it iterative and so on..? The only solution would be to replace this operator by a superfluous task and creating two additional abstraction levels in the process.

So the model was semantically altered simply to suit the notation needs but usually the notation and the domain model have different meta-models and more precisely the domain-model should not worry too much about its presentation (a domain-model can be represented in different ways which might be a future add-on to Hamsters; the aim is decoupling the model from its representation). That was a general rule argument but let's focus more on Hamsters case. In addition to doubling the decomposition levels inside the domain model we should notice how the *OperatorNode* shares various features with the *Task* in a way that makes it better to be modeled as a special type of Task rather than a semantically different element: Operator. This is the founding idea behind our adopted solution which is presented next.

4.2.2.2 Phantom Tasks

In order to solve this problem our solution need not to alter the domain meta-model in a way that changes its semantics. Our meta-model will be always a hierarchy of tasks no matter the notation is. Strictly speaking, it is the notation's job to adapt to our model and not the opposite. We need this total abstraction with the notation. This explains why we do not have an X and Y attributes in our elements to specify where they are positioned inside the graphical canvas: those attributes are defined inside the notation meta-model.

The alteration that we suggest is allowing tasks to be unnamed (i.e. anonymous or *phantom*). Anonymous tasks will have an operator as any other Task (in the end they are tasks). This way we have a task that does not require naming and description as its objective is not to present a real-life counterpart task but a transient grouping of tasks around a specific operator. In this case the notation will be simple to implement and remains valid as the notation meta-model can be represented graphically using an elliptical figure but at the same time allows it to benefit from the attributes and references it inherited from the superlative TaskNode classifier. This way we can define for instance that an

OperatorTask has to be repeated 10 times without requiring neither additional superfluous tasks neither doubling the abstraction levels.

Now, let's see how the model is represented in our pseudo-XML language following the new solution:

```
<taskmodel>
    <roottask name="T1" operator="|||">
        <subtask:task name="T2"/>
        <subtask:operatortask operator=">>">
            <subtask:task name="T3"/>
            <subtask:task name="T4"/>
        </subtask:operatortask>
    <roottask/>
</taskmodel>
```

Listing 4.2: Sample model structure using *phantom* tasks

In our new pseudo domain model, we have 2 abstraction levels instead of 3. The root task owns his children directly now without the superfluous "|||" in the previous example. The hierarchy is purely composed of tasks and their subtasks (the operator in this case can be seen as an attribute and not a node). Now, we can resolve the issue of adding some features like iteration to our operators easily (line 4 in listing 4.3) while designing our model graphically. The domain model will offer a seamless integration, our pseudo-model will be (although in the real domain model it will be containment but in either ways it is a matter of setting a reference):

```
1 <taskmodel>
2     <roottask name="T1" operator="|||">
3         <subtask:task name="T2"/>
4         <subtask:operatortask operator=">>" iteration="x10">
5             <subtask:task name="T3"/>
6             <subtask:task name="T4"/>
7         </subtask:operatortask>
8     <roottask/>
9 </taskmodel>
```

Listing 4.3: Sample model structure using *phantom* tasks with iteration

4.2.3 Tasks Flow

4.2.3.1 Errors in Task flows

A task flow is the series of execution a set of tasks follow in order to achieve a higher level task which is related to a well defined goal. This execution can be composed of multiple tasks that are inter-related. As most Task Models, Hamsters rely on temporal operators to define how a set of subtasks should be executed. In the easiest case, it could be simply a sequence and in complex ones a synchronized parallel execution. Another interesting feature of task flows is their ability to carry information. One task can provide a message to another task. In this section we will discuss parallel execution and synchronization because of its complexity compared to simpler executions. But first we will take a look how Hamsters models errors through two different Task Flows it supports:

Normal Flow This is an error-free flow of tasks. Usually this flow represents the ideal path that the user should follow in order to execute his/her task. Most existing Task Models supports only this type of flow.

Exceptional Flow Hamsters provides a direct support to model error-prone tasks. When a task raises an error, Hamsters will try to find if the analyst had provided any exceptional flow definitions and proceed along. Errors are usually encoded in the form of conditions attached to each task.

4.2.3.2 Parallel execution

Hamsters define two major parallel execution operators. The first simply indicates that both tasks can be ran in parallel and their constituent actions can be executed in an arbitrary order. The second allows the actions to be carried in parallel but restrain them to a set of synchronizations. The latter type of parallelism is usually defined at higher level parallel tasks. When those tasks are broken-down into smaller tasks, the way they executed in parallel can be more complex as one of them would wait for a specific action to be executed in the other in order to continue (logical ordering or waiting for an input...). Hamsters support synchronizations between tasks by allowing the analyst to create synchronizers between subtasks. This synchronizer would block the execution of one task and resume it only if the other task has started or finished execution depending on its definition.

4.3 Roles and Objects

4.3.1 Roles

Hamsters define various types of Tasks. Those tasks are classified rather based on the nature of the task. Most previous models classify tasks according to their executant. In Hamsters this remains true but we opted to provide a more fine-grained task types in order to make richer and more precise models. The model defines four major task types: Abstract, System, Interaction and User. Abstract tasks are usually present in higher abstraction levels of the model. Their abstraction came essentially from the fact that we do not have enough details to associate them with a specific type. The main reason behind this lack in details is that carrying this task requires different agents to collaborate (mix of abstract, system, interaction and user tasks). System tasks are simply tasks performed by the system, usually these tasks are not detailed by the analyst and seen as black boxes. User tasks are used by analysts to indicate that a specific task is performed by the user. Finally, Interactive tasks are a special type of Abstract tasks which mixes only system and user tasks involving interaction mainly input and output actions.

Hamsters, being a Task Model, should provide additional details about users. In particular, it should allow the analyst to be more specific about who is performing the task. This has many advantages, mainly task allocation and conflict detections (see 2.4.4 on page 26). Each Task Model in Hamsters is associated with a well defined role which can in turn have multiple models describing the different tasks it can perform. Roles are also mandatory if we want to create collaborative models; in this case the situation can get a bit more complex (see 5.2 on page 66 about the concept of *Actor*). One final interesting feature that Hamsters provides is allowing inheritance when defining roles, thus enabling analysts to represent complex organizational structures of roles. This property can be very useful for other modeling purposes too as we will see later.

4.3.2 Objects

In Hamsters *Objects* represent the world through their states. Tasks are defined to execute a set of actions that aims basically at manipulating objects and consequently the world. As in K-MAD, *Objects* are considered only second to tasks in the model. Our model allows analysts to provide a well defined formal description of the world by defining the different Objects that are found there. For the moment, Hamsters only defines the attributes of an object which together defines a specific state of that object and in turn the world. What is lacking is the concept of Object methods. Although the concept might be interesting, we think that introducing actions to objects conflicts with our Task-Oriented model. In addition most of these actions are usually performed by the system and consequently do not provide added value to our model.

One interesting challenge we faced when trying to include objects in our model is the *Class* vs. *Instance* problem. If objects are defined as classes, than our model should define a registry of Object-classes. Those classes can be used in different models to create different instances. If we consider an Object always as an instance, its definition should be provided when it is added to the model. The first solution would require including two concepts: Class and Object, while the second would require only one which can be called simply an Object. We opted for the first solution as the first solution would lead us to define a UML class-diagram like model inside Hamsters which can (1) shift the analyst attention from tasks to defining classes, (2) requires a set of additional elements in the meta-model in order to support this (basically consists of including a full or partial version of the UML class diagram meta-model). If different objects of the same type are used in different places, the analyst can rely more on the usability of the tool (mainly using copy and paste).

Chapter 5

Hamsters Meta-Model

5.1 Introduction

After giving some important foundation details about Hamsters, this section goes further by giving a more formalized description of all the introduced concepts fine-grained with additional details. These information were encoded inside a meta-model which we will be detailed in this chapter. But, first we will present some other foundational constructs for the Hamsters Meta-Model which are required to understand the meta-model later.

While designing Hamsters meta-model, we identified different levels in creating a Task Model. By level we mean the analysis frame or frontiers. Usually, task models are modeled for a particular *role* doing a particular *task* as you might notice so far. This particularity provides us with a well framed model and some abstraction (see 4.1.1.1 on page 53), but at the same time put some interesting questions on the table. What about relationships relating Tasks of the same role defined in different models, or moreover what about relationships linking tasks from different roles (i.e. Collaborative Tasks).

In our meta-model, we took into consideration these different levels of framing in task modeling and we make them an integral part of it. By not limiting the meta-model to only how constituent elements inside a conventional task model relate, we may achieve better consistency and organization among task models at higher levels. Before proceeding, these concepts described in our meta-model do not concern the task analysis and/or task modeling process; it does not define how the analyst should perform his analysis neither what process should he or she follows. To make it more clear, it is about structural organization and consistency and does not concern any process or global traceability. Our meta model defines three main task modeling levels:

1. Task Analysis Level

2. Collaborative Task Level

3. Task Model Level

We will follow a bottom-top approach in defining and detailing these levels in the following subsections (starting from the most specific).

5.2 Hamsters Modeling Levels

5.2.1 Task Model Level

The third level concerns the Task Model (TM) itself with the *conventional* understanding of it; as described by most models. Task analysts will work on this level of modeling most of the time as it is the most details-demanding and challenging to build in the process. A primary purpose of a task model is describing how an already defined *role* performs a *task* put into question to achieve a well defined *goal*. The answer to this question lies in decomposing this task; typically by subdividing it into smaller manageable and understandable subtasks (see 4.1.1.2 on page 54). Another important feature that TM provides is specifying how these subtasks are related mainly in time (usually relates tasks at the same level of abstraction but there are some cases where more sophisticated relationships are required between different tasks from different abstraction levels; see 4.1.3 on page 56 and 4.1.2 on page 55 for additional details). As noted earlier this type of task modeling is framed by two constraints:

Only one role is implicated in performing the task Although, it is worth mentioning that technically speaking not all described subtasks are performed by it. There are some predefined agents that could perform these tasks. Actually there is a one major agent: the System (any task performed by the System). Other tasks are said to be performed *interactively* but do not have detailed subtasks to state in a better precision who did what (see 4.3.1 on page 63). In the bottom line, it is going to be performed thanks to collaboration between the role and the predefined agent System, but to avoid confusion we better define it as another type of agents: "Interactive" (these tasks can perform only interactive actions: input, output...). For example the task "Print Document" is performed *interactively* (the user presses the print button than the system outputs the printed document). We mentioned the subtasks just to explain the collaboration but in the task model they can be omitted; mainly because they are not relevant to the Task Analysis domain.

Only one major task is described by this model We used the ambiguous adjective major intentionally because it is the job of the analyst to define which tasks are considered major and thus require TMs to describe them. This constraint frames the analysis problem and focus on this particular task.

Both constraints have consequences on our modeling. We will start by the positive ones:

1. Help the analyst focus on one role (see 4.1.1.1 on page 53).

2. Help the analyst focus on one task.

3. Implicit assignments of tasks: we know which role will perform which task (see 2.4.4 on page 26 and 4.3.1 on page 63).

4. It results not only in a task-focused model but also a role-centered one , we can say a role-centered analysis too. (implying automatically User-Centered analysis).

5. Simple to interpret and analyze by automatic tools.

Among the disadvantages we can cite:

1. What about complex identified major tasks. Modeling a task can be cumbersome in complex environments and would require further framing of subtasks. In other words we cannot use one TM to fully model a complicated major task.

2. Limit the analysis scope which can be confusing for some types of tasks where inter-operations among tasks and collaborations among roles is a necessary to achieve certain goals.

3. What about tasks which can be performed by multiple roles. In this case the analyst is required to model the same task for each role.

Our meta-model can tackle easily those disadvantages by using the already introduced concept (Modeling Level) and second by providing new elements or altering existing ones to bypass these limitations. Point 2 is easy to refute by our second Modeling Level: Collaborative Task; explained more below. Point 3 can be resolved through the introduction of a better structural feature called *Actor* and by enabling inheritance among roles. An Actor is a special agent that can have multiple roles. It is better to abstract actors from roles and make them modeled separately as they are more likely to change in the future. This separation avoids re-modeling tasks each time a new organizational hierarchy structure is imposed which is common nowadays.

May be the most challenging argument is the first one. In order to solve it we introduced a new relationship among TMs which is simply: TM could have zero or more sub-TMs (in reality, they reference one or more tasks defined in that sub-TM and not limited to the root task; this will be explained more when we describe the TaskReference element; to make things easier we opted for this explanation for now). So in this case we have a TM that contains some subtasks described in other TMs (i.e. referencing another TM). This way we can always have a manageable TM and avoid complex models. This property provides us the right time to mention two distinctive types of TMs:

Basic Task Models They are the basic building block of the whole global model. A basic model does not have any sub-TM. All of its subtasks are defined and modeled locally.

Composite Task Model We can consider them as a higher level task models but what really makes them special is that they reference other sub-TMs. One important constraint arises in this case: any reference to any sub-TM can be nothing but an atomic subtask in its referencing TM. Beside the abstraction gain we have by applying this constraint, it keeps the model of the referenced TM independent from its referencing TM (so any modification to that task can be only made in one central location) and also avoids confusion (defining additional subtask for a sub-TM reference raises the question: how these are related to the original model?). One last thing to mention is that Composite TMs are not limited to reference only Basic TMs but can reference other Composite TMs giving the precondition that each of these TMs does not reference them in any way (if allowed this will create a vicious circle).

Notice that any Basic or Composite TM can be referenced by more than one Composite TM. This property open the door to an important feature of our meta-model which is reusability. You can describe a specific task once and include it in multiple composite models with a simple reference! Nevertheless, there is an important contradiction that will arise. We said that a TM concerns one and only one role, the question if we allow reusability through referencing what will happen if a Composite TM performed by role A references another TM performed by role B. A simple answer would be to consider such references illegal and eventually ban them, but in real life we have many cases where smaller tasks are shared among multiple roles. So limiting reusability to one role would not be really beneficial without enabling it between roles. Our solution to this problem is the use of generic roles (specialized roles inherits all tasks of their parents). We introduced also a special role called *Anonymous* as the most generic role (all roles inherits from it directly or indirectly). Anonymous TMs can be performed by any role referencing them. Again one can argue that if we allow Anonymous roles we cannot optimize tasks and analyze them correctly because we have no idea who is performing them. This leads us to how a composite TM can reuse an anonymous TM? We propose two modes of reusing Anonymous TM to solve this issue:

By reference where the anonymous task is not affected by its referencing TM. This preserves the reference integrity and allows a long-term efficient reusability.

By replication where the reference replicates the anonymous task content but cannot guarantee full future reference integrity. Replication could be performed locally or results in a new sub-TM when we are dealing with a complex TM.

When it comes to how the task should perform or how it should be executed we can identify two ways:

Inclusion The task is included inside the model and fully integrates with it. In this type of referencing, the analyst should have access to the internals of this task. The task in this case needs to be aware of its environment.

Extension The task here extends an existing model by providing additional details. This type of tasks tends usually to be optional and does not depend on the environment. These types of tasks are included in a *read-only* mode.

There is another kind of reusability worth mentioning: *in-model* reusability. In this kind of reusability we reuse a task inside the same model more than once. This is completely allowed in our meta-model but as any reference it has to be atomic. This implies that the reused task needs to be described at least once than referenced by other tasks. Where to describe the actual task is a decision that needs to be made by the analyst (sometimes it makes sense to have it described in one location and referenced in others).

5.2.2 Collaborative Task Level

The second modeling level concerns collaborative tasks modeling. A collaborative task is a special task performed by more than one role. This level of modeling is very intuitive to understand at first. To construct a Collaborative Task Model (CTM), we need first to construct our role-oriented TMs than reference them in our newly created CTM. In this regard, CTMs share some common properties with Composite TMs including mainly the inherited constraint "references to other TMs must be atomic". Actually this can be a serious problem for collaborative tasks as they are going always to have one task abstraction-level; the root collaborative task will be eligible only to reference other TMs which cannot be subdivided further as they had already been detailed in their respective real models.

To be more precise, we can construct new subtasks and consequently new abstraction levels but those tasks can only be performed by the predefined *System* agent. The problem with the remaining task types (User, Interactive...) is that they cannot be inserted due to the fact that they need a role which is missing from our CTM in the first place. Any CTM that will include tasks other than what the referenced TMs describe will need a role in order to function. Thus we allowed CTMs to have an optional role which is played most of the time by a role of a higher organizational level whose responsibility, usually, lies in enabling collaboration between different actors' instances. Notice we used the term *actor* and *instance* instead of simply roles. CTM have a special attribute: role or actor instances. This enable collaboration or task partitioning not only among roles but among instances of these roles which is what happens in real life most of the time. For example a leader L commands A who holds the role R1 to perform the task T1.1 (which can be performed by instances of role R1). Imagine that the leader will need to perform another T1.1 in parallel so he will likely need another agent of R1. Role instances and other TM elements instances will be discussed in details, including the above situation, in the Simulation section (see 5.4 on page 81). However before proceeding, an important property should be mentioned: the same CTM and more generally Composite TMs can reference another TM more than once.

5.2.3 Task Analysis Level

The highest level of modeling is the Task Analysis Model (TAM). The TAM encloses all elements belonging to other levels of modeling. It can be considered like the modeling project. For instance when an analyst starts analyzing a specific case like Air Traffic Control, he begins by creating the Task Analysis Model for this particular application. Later on, he works on the building blocks (other low level models). This project-level container allows better organization as said earlier and allows us to have a centralized way of defining, manipulating and storing shared properties and elements. The first example that came to the mind is the roles; the TAM allows us to manage all the roles related to our project in one place; of course all other common elements need to be moved to this level.

Another interesting feature of TAMs is their ability to contain other TAMs. This is particularly very useful in modeling very complex systems by allowing analysts to divide the system into smaller components or subsystems. Furthermore, when dividing the system into smaller subsystems, we can run special automatic or even manual analysis in a more accurate way. For instance, run a task query within a subsystem scope.

TAMs support also what we call registries. A *registry* is simply a global container which allows different components to access a set of elements belonging to the same type. For some elements it plays the role of the storage container. For example roles and actors will be stored in their respective registries as they are not dependent on one TM or CTM (a role can have many TMs). Registries provide different facilities including performing an action that will affect all elements of the same type (adding prefix to all tasks) or simply allows some queries or statistics. This is especially true for the Tasks registry where we register all tasks defined by this TAM. Another feature of registries is its simple interface to communicate with other TAMs. The Tasks registry alone (*alone* hear means provide one external interface!) for instance can export tasks easily to be copied or reused through referencing in other TAMs (this is especially true for task patterns).

5.3 Hamsters Meta-Model Elements

In this section we will try to give a description of Hamster's meta-model elements. These elements are basically partitioned into two types : relationships and elements. We chose to model Hamsters into two diagrams to improve readability and to separate concerns. The first diagram (see figure 5.1 on the facing page) gives an overview of Hamsters meta-model from a higher level, basically exposing Hamsters modeling levels.

The second diagram (see figure 5.2 on page 72) describes our *conceptual* meta-model by detailing the *TaskModel* element content (found in the first diagram and the lowest modeling level in Hamsters). We call it *conceptual* because we will alter this conceptual version progressively to support features required by simulation, notation and the implementation. What this meta-model represents is the essence of our task model (the final result of the model in its static form would be viewed from this perspective for all automated processing of the model). Most attributes were omitted in order to focus on the structure of the meta-model: constituent elements and their relationships. Most of the terminology we are going to use to describe our model is based on the Eclipse Modeling Framework.

There is also a third diagram that follows (see figure 5.3 on page 73). It builds on the conceptual meta-model and integrates some implementation-aware modifications. This meta-model plays the role of a bridge between the concrete model representation and its abstract one. The conceptual model can be obtained simply by running a model transformation script.

5.3.1 TaskAnalysisModel and CollaborativeTask

These elements are mainly used to describe the first and second modeling levels of Hamsters (see section 5.2 on page 66 for details and figure 5.1 on the facing page for their placement inside the meta-model).

TaskAnalysisModel It represents the global task modeling project. It contains properties and attributes that are related to our project (project name, organization, version. . .) It contains also a set of TaskModels. In addition, it encloses four major registries (one for Roles, a second for Actors, a third for Tasks, and a fourth for Objects).

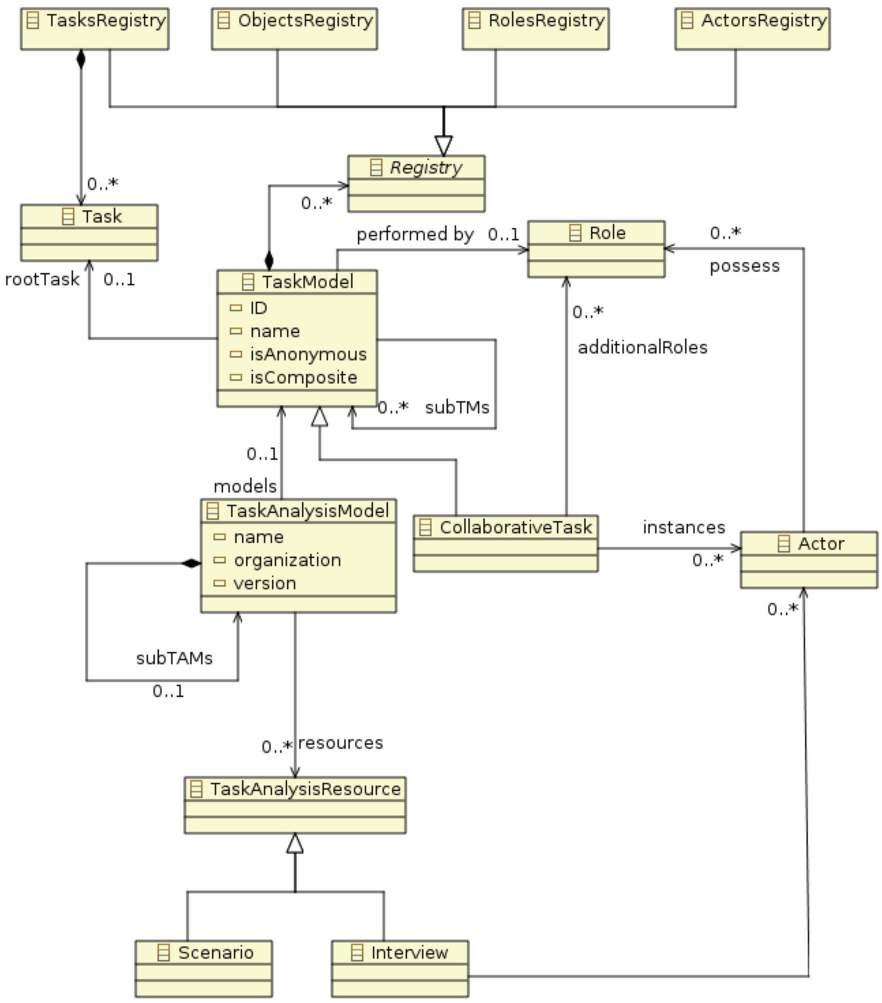


Figure 5.1: Hamsters Higher Level Meta-Model

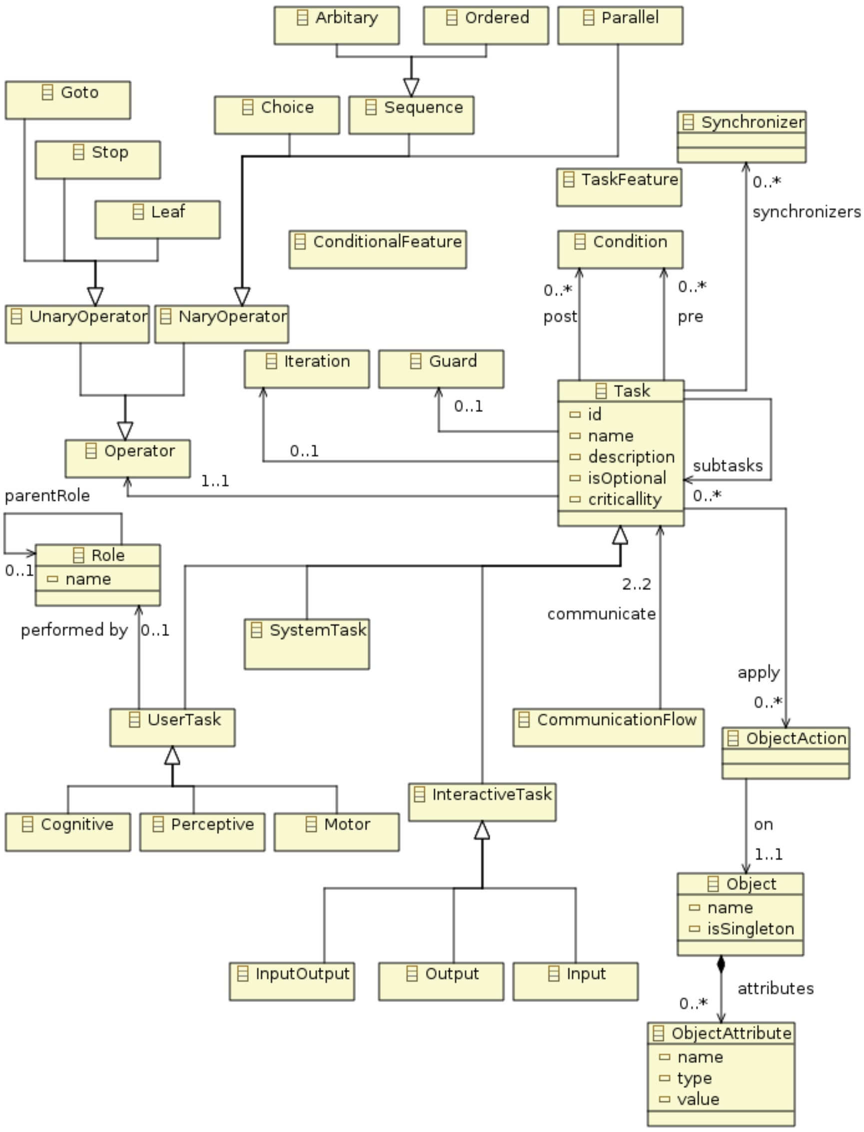


Figure 5.2: Hamsters Conceptual Meta-Model

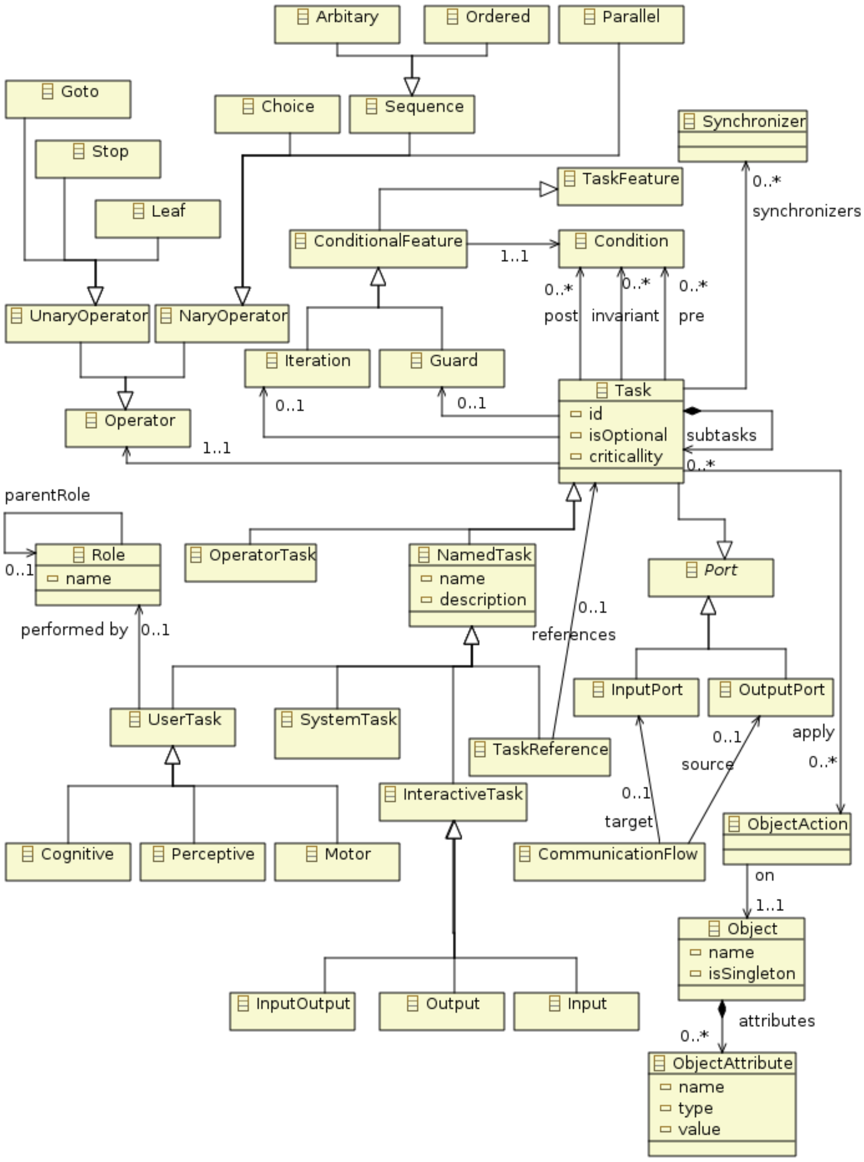


Figure 5.3: Hamsters Implementation-Aware meta-model

CollaborativeTask A specialized classifier of *TaskModel* where task references are allowed for different roles (the reference *additionalRoles* in the meta-model is a derived reference; can be computed). Remember a normal TaskModel can reference either TMs having the same role or marked as Anonymous. Another interesting reference is *instances*; it is used to distinguish tasks that are meant to be performed by the same role but with different actors.

5.3.2 Registries

The following classifiers are used to define registry related elements:

Registry The super type of all registries, it holds all common attributes and operations common to all registries.

RolesRegistry This registry contains all the roles defined inside its containing TAM. Roles can be added directly into the *RolesModel* or through creating a new TM and assigning it a role that does not exist. The *RolesModel* is an utility element allowing analysts to describe roles without necessary creating a task model. This is especially useful for existing systems.

ActorsRegistry This registry contains all the actors defined inside its containing TAM. An Actor corresponds to a real agent (instance of role). Actors can belong to more than one role (reflecting real-life organization). Actors are modeled inside the *ActorsModel* which plays the same role as *RolesModel*.

TasksRegistry This registry contains all the tasks defined inside its containing TAM. This registry is useful for different purposes (see above).

ObjectsRegistry This registry serves as a central container for all Objects used in our TaskModels.

Registries are used to store elements that are global in nature or in some cases to create an index for some elements. Among global entities we can include the Roles and Actors registries. As for an index registry we can cite the ObjectsRegistry.

5.3.3 TaskModel

It represents task models as described in 5.2.1. Note that we did not create a specific type for Composite TMs on the other hand. A Composite TM is nothing but a TM which holds at least one reference to another TM or TM's tasks. Nevertheless, we defined a special *derived* attribute called "isComposite" for the convenience (for example performing the right interaction when copying, moving, deleting, can be useful for filtering too etc.). The task model has the following additional non-derived attributes :

ID This TAM unique identifier (will be generated automatically but analysts can always override it given they respect the ID naming constraints.

Name a human readable name for this task model. Useful for the notation and documentation.

Description: a description of this task model, same as Name, useful mainly for the documentation.

The task model has the following references:

role Determines which role should perform this task model. If no role was specified, the task will be assigned to the predefined *Anonymous* role.

rootTask The root task of this model as every task describes one and only one task which could contain itself subtasks (see below for the a discussion about rootTask vs. TaskModel).

Every task is performed by a role (if not specified the default Anonymous role will be selected). To define roles we use the Role classifier:

Role It defines a role in the system. A Role can have a parent role inheriting all its tasks in addition to its properties (using the *parentRole* reference). Hamsters define a default role named *Anonymous* that will be used whenever the analyst does not provide a custom role. Every task performed by this predefined role can be reused anywhere inside the project.

Before moving to the next section, we would like to argument our choice of having a *rootTask* inside a *TaskModel*. The TaskModel is meant to provide a detailed description of a *major* Task. For every task model we will have one and only one root task. The question is why don't we consider simply that a TaskModel is nothing but a task? The primarily answer is providing a specific type to represent the Task Model Level, and to provide some additional information related only to the TaskModel. Moreover, in technical terms, this choice makes implementing diagram referencing and partitioning easier for our model. However, Thanks to multiple-inheritance, in our implementation meta-model we considered making the TaskModel inherits from the *Task* classifier. This way we preserve two important features:

1. TaskModel is semantically an independent element.
2. TaskModel can be use in diagram partitioning as it is a *Task* itself.

5.3.4 Tasks

This section will describe the most important elements of our meta-model–tasks. In Hamsters tasks are not typed using an attribute as most task models. The task has its own class allowing us to add relevant attributes and actions depending on its type. We followed previous models in dividing tasks into four major types: Abstract, User, System and Interactive. However we did not stop there, we went identifying more specialized types for User and Interactive tasks (specialized types of System tasks are not necessary because they are always seen as black boxes from the Task Modeling perspective).

The following list defines all used classifiers to denote a task in our model:

Task It is a classifier used as a super-type to define any special task type. The Task element has the following attributes and references:

ID	A unique identifier for this task.

isOptional Determines whether this task is optional or not.

criticality Determines the criticality level of the task. This is an integer value, 0 means neutral, the greater the value, the more critical the task becomes.

subtasks Every Task can have zero or more children subtasks. In our conceptual meta-model which supports heterarchies natively, a task can belong to multiple parent-tasks. However, in our implementation meta-model, every task may have either zero or *one and only one* parent (this may seem at first as we are breaking the reusability feature but using another magical element called *TaskReference* reusability will be indeed retained; see below) .

preconditions, postconditions Defines the set of conditions related to this Task. See *Condition* description for details.

operator Each Task has at least one operator. This operator will define the execution flow of underlying tasks. See *Operator* description below for more details.

Set of features, mainly we can cite:

iteration: a reference to an Iteration which describes how this *Task* iterates.

guard: a reference to a Guard specifying when this *Task* could be executed (see Precondition vs. Guard discussion below for differences).

synchronizers: Synchronizations related to this Task. Only descendants of Parallel tasks can have synchronizations. Synchronizations and parallel tasks are a large topic, you can consult Parallel Task Modeling section for further details (see 4.2.3.2 on page 63). This reference is navigable in both directions.

Our meta-model has the following specialization classifiers for *Task*:

SystemTask Describes a task performed by the system.

UserTask A classifier for abstract tasks that are performed solely by the user. An interesting reference only UserTask possesses is the possibility to associate it with a role. By default all tasks are performed by the default role associated with the containing TaskModel . The analyst can override this behavior by specifying in a more accurate way which role exactly performs the task. However, this role has to be an ancestor of the default role (through the *parentRole* reference). UserTask has the following specialization classifiers (note: any subtask of UserTask has to be a direct instance of UserTask or one of its specializations):

MotorTask A physical task or activity.

CognitiveTask A cognitive activity (calculation, decision making, analysis. . .)

PerceptiveTask The user perceives something (seeing a plane for instance in ATC).

InteractiveTask: A classifier for interactive tasks; tasks that are performed interactively between the User and the System. By interaction we mean exchange like in request-feedback or feedback-reaction. The used naming terminology describes specializations from the user's perspective (as an Input action for the user is seen as an Output action from the System perspective). It has the following specialization classifiers (note: any subtask of InteractiveTask has to be a direct instance of InteractiveTask, or one of its specializations, see the end of this section for differences between InteractiveTask and AbstractTask elements).

InputTask A task where the user provides input to the System.

OutputTask A task where the system provides an output to the User.

InputOutputTask A mix of both but in an atomic way (see the end of this section for differences between InputOutputTask and. InteractiveTask).

It should be noted, that the use of specialized tasks depends on the analyst. This increases Hamsters flexibility by supporting different level of details. We can use Hamsters to describe high level abstract tasks, as to detail low level concrete tasks (like KLM).

In our conceptual meta-model, we do not deal directly with the problem of integrating operators inside the task model which was discussed in 4.2.2 on page 59. But in our implementation meta-model, we distinguish two major types of *Task* to support this feature:

NamedTask has a *name* and maps to a semantically existing task in real world. Any instance of this classifier or its descendants are usually seen as the real constituent of task models. Its direct instances represent Abstract tasks.

OperatorTask It inherits from *Task* but does not require a name (thus the other naming alternatives *Anonymous/Phantom Tasks*). They are special in way that they do not have a direct real-life counterpart. This kind of tasks is useful to model *convenience* task nodes which are needed to group a set of subtasks around a specific operator eliminating superfluous named tasks and operators priority problems. What makes them special is that they do not require a name so they can have a special representation in our notation model (this choice raised an interesting discussion while designing the meta-model, see OperatorTask vs. Operator for a comparison in 4.2.2 on page 59).

Another interesting element that merits mentioning but found only in our implementation meta-model is:

TaskReference This classifier was introduced to solve the paradox that you might notice earlier: *"How could a task be reusable while it needs to have only one parent at anytime?"*. The answer is using a special adapter classifier called TaskReference. This classifier links to its target task through its *referencedTask* reference. Another benefit for adopting such approach is making the subtasking constraint (any reference has to be atomic) easy to implement. One additional gain is that TaskReference can be viewed as a Proxy pattern too. This is will help the implementation to be more efficient and boost the performance for complicated models as we are not required to load the actual referenced task, will make lazy loading easier to implement.

Before closing this section, we would like to eliminate some ambiguities related to the definitions of *AbstractTask, InteractiveTask* and *InputOutputTask*. Since an *AbstractTask* has a set of *subtasks* performed either by the *User* or by the *System*, Why not consider it simply as an InteractiveTask? Or remove the InteractiveTask as it is representing the same thing as an AbstractTask. The question is How an InteractiveTask is different from an AbstractTask? In fact, AbstractTask plays two roles: it is the super-type of all named tasks and it is the class that can be subtasked into any other type of tasks. So it has a broader definition as the adjective Abstract implies already. InteractiveTask on the other hand is a more specialized version where a well semantically defined tasks are classified as Interactive. An InteractiveTask is a task where we can witness a direct interaction between the User and the System so in other terms it requires an interface, consequently an Input from the User and a feedback or an Output from the System. Additionally, InteractiveTasks are limited to subtask only tasks of type InputTask, OutputTask or InputOutputTask.

As for the InputOutputTask, it usually describes a task where the Input provided by the User and the Output provided by the System are strongly coupled that we prefer not to break them into subtasks. This description can be used to imply that all InputOutputTasks are orphans by definition. The provided description and argument above are strong but they do not really imply banning the InputOutputTask from subtasking. For instance an analyst could perform a KLM (Keystroke Level Modeling) so he will need to decompose further every InputOutputTask but at the same time preserve the property of coupling. We prefer giving more freedom to the analyst in this case rather than restraining it.

5.3.5 Conditions

Conditions are very important in order to verify and validate the task model execution. In particular, they play a key-role when modeling critical-interactive systems. In Hamsters conditions are modeled by instantiating the *Condition* classifier:

Condition A classifier to model conditions. Conditions are like test units or verification flags that might signal a problem if anything wrong happens when it should not be. When attached to a *Task*, Condition can be added to different references:

- **Precondition** Conditions of this type are validated before proceeding the execution of the Task. If any condition fails, the task will not execute and a default *TaskPreconditonFail* exception will be raised.
- **Postcondition** Same as Precondition but differs in when to verify. Postconditions are validated after the task execution. If any condition fails, a default *TaskPostconditonFail* exception will be raised.
- **Invariant** Same as above but will validate before and after execution. Another particularity of this type is that its containing task will

validate it before and after the execution of any subtask. This is equivalent somehow to the Invariant loop principal in algorithms (the condition needs to remain valid at all times).

Before proceeding to next meta-model element, you should know that the analyst can override the default Condition behavior of simply raising a *ConditionFail* exception by defining the next exceptional flow in this case. Each condition can have its own next exceptional flow making it possible to respond properly to different invalid conditions. You can have more details about exceptional flows and task exceptions in the Simulation section 5.4 on page 81.

When it comes to conditions and their relationships to tasks, one can argue: what is the difference between a Task's *Guard* and a Task's *Precondition*? As we might use preconditions to constraint the task execution given if a precondition is not valid, the task will never execute.

Actually, there is a major difference between a Precondition and a Guard. A Guard simply *guards* the task, in other words it will allow the task to execute only when it evaluates to TRUE. Otherwise, it will pass the execution flow to next possible task. As for the precondition, it is true that it constraints the task execution in the same way but the outcome is different in both cases (TRUE or FALSE). When it evaluates to TRUE, there is no guarantee that the task will execute as we could have other preconditions that will prevent the execution anyway. If it evaluates to FALSE, the task would not simply abort the execution but will raise an exception. That marks clearly the difference between passing the execution (Guard) and halting it or diverging to an exceptional execution flow (precondition).

Talking about Conditions, we considered the following additional question: Why a Task has a 0..1 multiplicity for the Guard (allowing maximum one Guard) but it has 0..* multiplicity for Pre-conditions? The answer to this question is also related to how the behavior resulting from these are different. When a condition raises an exception we would better give more details about this exception so we could proceed to the right exceptional flow or provide a specific behavior. In the case of Guard, we do not need such feature. We can specify multiple conditions using the AND operator (possibly using a Task Constraint Language TCL; see Prospects).

5.3.6 Operators

Operator A super-type for different types of operators. There are two main operator groups:

Unary operators They are special operators that operate on one operand which is usually their containing task (which can be an OperatorTask). We can cite the following unary operators:

Stop	Halts the whole task execution. Use Stop operator when everything is done and no need to go further in execution. A Stop is a clean way to end the task execution.
Leaf	This operator is a *null* operator (does nothing). It is used to indicate that the containing task is considered atomic for the analyst.

Goto	An operator that jumps the execution to another task.
Call	An operator that calls another task in the hierarchy, the environment state will be preserved (objects will not reset). If the operator is calling an ancestor, this ancestor will be considered automatically a recursive task. Any task that contains a Call operator and is calling its ancestor has to define a Guard or it has to be enabled conditionally by other tasks.
Error	The same as Stop but signals an abnormal exit.
Exception	Raises a task exception (see 4.2.3.1 on page 62 and 5.4.2.2 on page 84).

N-ary operators They are the most commonly used type of operators. They define how subtasks should be executed; the task flow (which one, in sequence or in parallel. . .) An N-ary operator must have at least two operand subtasks. Among the most used n-ary operators we can cite:

Sequence	It executes the subtasks in sequence. Two behaviors are possible: following the given order (Enabling) and arbitrary (Order Independence).
Choice	Only one subtask will be executed. The chosen task will be either selected explicitly by the user (in non-deterministic cases), selected by the execution environment automatically or finally will select the first task that returns true for its Guard will be chosen.
Parallel	It executes subtasks in parallel. Parallel operators can define zero or several synchronizations. Parallel tasks and task synchronization are discussed in a special section within Task Flows as they represent some important particularities with many effects on the task model (this is especially true for critical tasks), see 4.2.3.2 on page 63 for more details.

5.3.7 Objects and Communication Flows

It is agreed that Task Models are task-focused, nevertheless objects still remain important to design a complete model. Objects in task models are the most generic element of information flow between tasks. A task can modify, produce and consume objects. That is why tasks need a method to allow communication between siblings other than temporal relationships which are attached at a higher level (parent task). To solve all these issues we define the following elements:

Object A classifier to represent objects in a generic way. It has a name and a set of attributes, each attribute has a name, a type and a value.

CommunicationFlow A classifier to abstract any type of communication that can occur between two siblings (i.e. having the same parent). A CommunicationFlow can vehicle different types of information mainly Objects through its objects reference. In its implementation meta-model, a CommunicationFlow will always target an InputPort and it will be always issued by an OutputPort. It is contained in its source port (i.e. the OutputPort sending it).

ObjectAction This classifier allows the analyst to define actions that the task can undertake to manipulate objects and thus change the world state. Additional specialized actions can be defined to alter objects in a well defined way. For example:

UpdateAttribute It changes the value of an attribute inside the object.

CloneObject It creates a copy of an Object.

The implementation meta-model appends another element to support ports:

Port A port is like a communication channel that has some specific properties (perception bandwidth for instance). Each Task can have one or more ports which could be helpful in case we have multiple communication channels between siblings. There are two types of ports: (1) *InputPort* acts like a reader of external information flow, (2) *OutputPort* acts like a writer to send information through its associated task flows.

As you might notice in the implementation meta-model, the *Task* classifier already inherits from InputPort and OutputPort (shown as from *Port* only in the diagram for readability purposes). This inheritance will enable our Task to act like a Port thus allowing CommunicationFlows to target the Task directly without forcing it to pass through a port. This default behavior can save the model many superfluous elements and will allow the user to design his model with a larger freedom as he might not need Ports.

5.4 Task Simulation in Hamsters

In this section, we give some details about the way Hamsters supports task simulation. The section will start by describing a modified version of Hamsters meta-model (only relevant elements will be shown). Next we will detail how Hamsters runs task simulation. The final subsection will give practical details about task flows, mainly how the simulation supports exceptional flow.

5.4.1 Simulation Extensions to the Meta-Model

Simulation lies in the core of almost any task model. Hamsters adds support for simulation by extending its core meta-model. Until now, the meta-model was static in nature and provides only information about the various relationships and attributes of each element. Simulation will alter this static model to add some behaviors to it by appending actions or methods that can be carried by active elements (elements exposing behaviors). The figure 5.4 on the following page shows again Hamsters conceptual meta-model with some additional extensions carried by task simulation. The table 5.1 on page 83 gives a short description for important methods of some simulation-core elements.

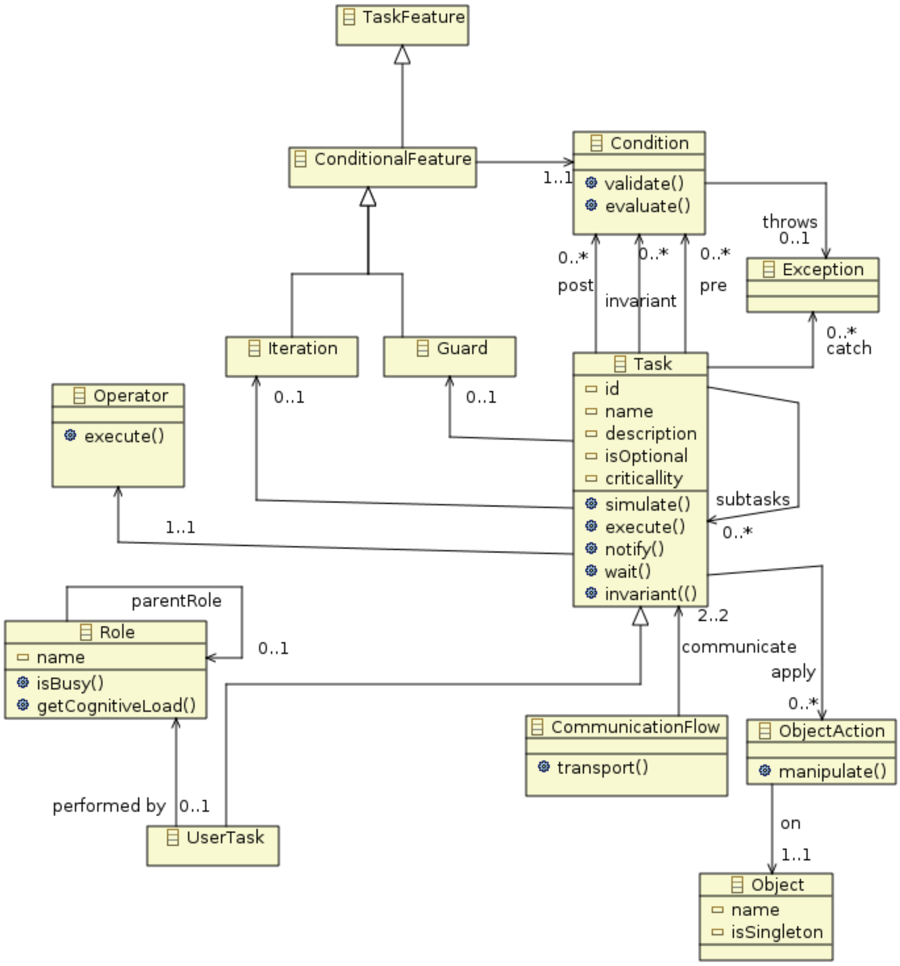


Figure 5.4: Hamsters Simulation Meta-Model

Classifier	Method	Description
Task	simulate	The main method that asks this task to start simulation. It handles itself the whole simulation process locally starting from this task.
	execute	Execute the task, basically it executes actions (mainly ObjectActions).
	invariant	Validate invariant conditions; will call parent task *invariant* method if any.
	wait	In parallel execution asks the task to wait for another task to start or to finish execution.
	notify	In parallel execution asks this task to resume execution.
ObjectAction	manipulate	Manipulates the associated object. This is an abstract method, specialized classifiers should define concrete implementations.
Condition	evaluate	Evaluate the condition definition. Returns TRUE or FALSE.
	validate	Calls evaluate and return an ErrorException when false, NULL otherwise.
Operator	execute	Execute operands according to this operator behavior. This is an abstract method, specialized operators should provide concrete implementations.
Role	isBusy	Indicates whether the role is busy doing something else or not.
	getCognitiveLoad	Example method to demonstrate Hamsters flexibility. This method can be implemented to calculate accumulating cognitive load.
CommunicationFlow	transport	Transports all objects to their destination.

Table 5.1: Important Simulation Methods

5.4.2 Principle of Hamsters Simulation

5.4.2.1 Simulation Execution

The simulation execution in Hamsters is launched by calling the root Task's *simulate()* method. This method will proceed in the ideal case (no exceptions) by:

1. Checking Task Guard. Return if guard evaluates to false, continue otherwise.
2. Checking all preconditions
3. Checking all invariant conditions of this task and if it has a parent tasks call its *invariant()* method.
4. Reading any objects sent using the CommunicationFlow if any.
5. Calling the attached operator *execute()* method and pass execution to the operator. The behavior depends on the operator type. For the case of n-ary operators, it will call the *simulate()* in a specific order, call (synchronous, asynchronous), ...
6. Execution resumes inside the task body. Now execute the task itself by running basically its ObjectActions to change world state.
7. Checking all postconditions.
8. Checking all invariant conditions of this task and if it has a parent tasks call its *invariant()* method.
9. Sending Objects using the CommunicationFlow if any.

Note that these steps will be the same for all subtasks resulting in complex recursive executions. You can find a simplified sequence diagram of Hamsters simulation principle in figure 5.5 on the facing page.

5.4.2.2 Exceptional Task Flows in Simulation

Hamsters deals with exceptional flows using the Exception object. Basically, it resembles to the *Exception* principle in some programming languages (Java, C++, ...). To model an exceptional flow, the analyst can choose among the following methods:

1. Create an explicit Task that raises exception every time its guard evaluates to true. For instance to verify a date and launch an exception if it is malformed. The analyst practically will proceed by adding a Task and attaching to it a guard (defining the condition to throw the exception) and attach to it the Exception unary operator.
2. Specify Exceptions when defining validation conditions (pre/post/invariant conditions), overriding the default ConditionFail exception. Whenever a condition evaluates to false, it will throw the defined Exception.

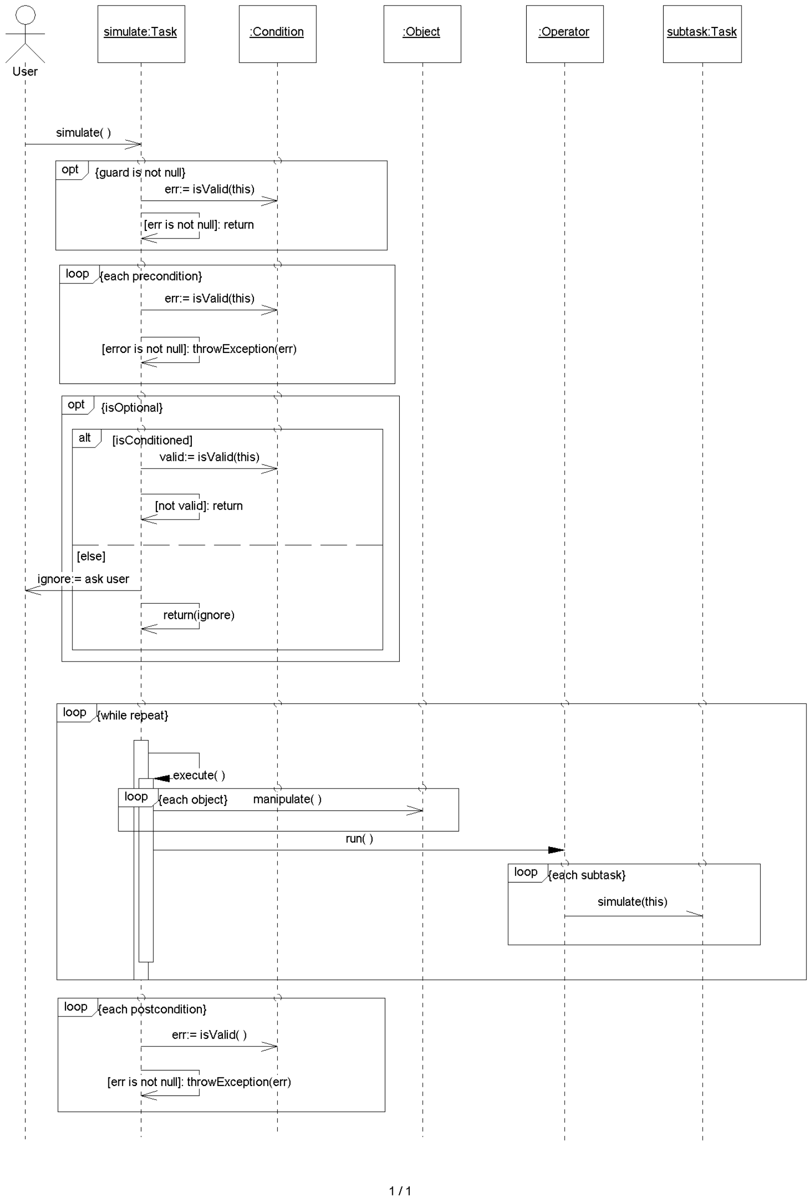


Figure 5.5: Hamsters Simulation Basic Sequence Diagram

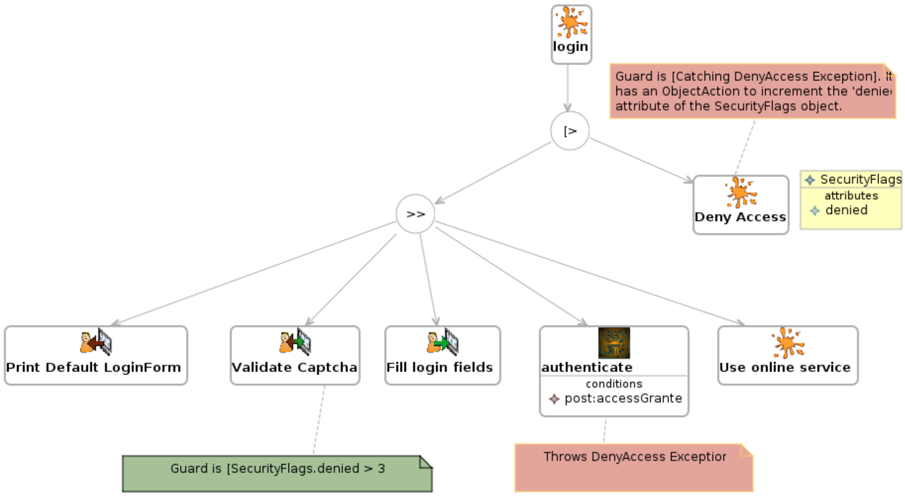


Figure 5.6: Example of using exceptional flow

Now after defining where exceptional flows can generate, the analyst needs to specify where to send the execution after. Hamsters enables this by allowing ancestor tasks to catch exceptions. This is done by defining a list of exceptions that any Task could deal with. The simulator will pass the execution to the first ancestor that catches the exception starting from the bottom. Note that there is a special disabling operator that has a different behavior by propogating the exception to its latter operands. To define a complete exceptional flow (series of tasks executed only in abnormal cases), the analyst can use a special guard that evaluates to true only with the presence of an Exception.

As an example, let's consider the case of a login task in web sites. The system allows a maximum of three trials, than it will resorts to use captcha to protect the system but increasing required cognitive load (see figure 5.6). To describe this, the analyst should add a guard to the captcha task. The system *authenticate* task defines a postcondition that throws an exception if *accessGranted* condition evaluates to false. After throwing the exception, only the *Deny Access* task will be able to catch it. This task is guarded so only when an exception of type DenyAccess is arisen, it would execute. The task defines one simple action that increments the *denied* attribute of the *SecurityFlags* object. After three trials, the *denied* attributes will be greater than 3 allowing the *Validate Captcha* task to execute.

Chapter 6

Model Notation

6.1 Modeling notation

6.1.1 Introduction

6.1.1.1 Purpose of defining notations

Notation is the language which we use to represent our models. Mapping models from a pure abstract form into a concrete one requires us to define a set of representation rules. Notations are used mainly for two purposes. The first is to find a way to express our models using more intuitive and concrete forms. The second is making our model human-friendly (i.e. readable).

Models are used to capture data about the modeled thing. However, this data now formalized into the model, is usually abstract and stored as information which can be processed only by its holder (human models are stored and processed by the brain, engineering models are usually stored and processed by computers). The notation for models is like language to human beings. It allows the model to be expressed in a unified fashion that can be understood by people. Notation is not only about readability but it is mandatory to create models. It allows the modeler input new data into the model using this same notation without dealing directly with its abstract or digital form.

Notation is not only necessary during model creation. It is also very useful in post-creation stages of the model. It is a key-requirement to communicate models. Using a notation, the engineer is able to show his model to different stakeholders or get it modified by another engineer; as in the ideal case the model speaks for itself through its notation. Simply, notations are the main medium models use to communicate with the external world. They are used to input data into the model, to represent data out of the model, and more importantly for communication and analysis purposes.

6.1.1.2 Importance of defining carefully the notation

Modeling is widely used in various domains, but no other discipline has sparked deep discussions lately more than computer science and especially Software Engineering. While modeling in its core gets a big attention from the community, its notation and language were put into second raw if not ignored. In fact, the cognitive effectiveness of notations, for instance, has been widely taken as an assumption and got accepted without any critical observation for a long time [Petre 1995]. If we look at the majority of models used to model Information Systems for example, we will find that they are not effective in communication, if not a source of trouble [Kimball 1995; Nordbotten and Crosby 1999]. The problem lies in the way we deal with notation definition. Different factors such as the crucial factor how we perceive things are not given any importance. Therefore, the process of defining a notation should be performed carefully and take into consideration a set of more formal methods or principles instead of relying on instinct and assumptions.

6.1.1.3 Notation types

In order to represent a model into a more concrete form, we can employ different types of notations. Those types usually differ in their readability and their expressive power. Of these notations we can cite the most widely known ones:

Textual This notation relies on text to represent the model. Its formality depends on the model's. In formal models, it has a well defined syntax. This representation can be very powerful to express mathematical and complex models but it tends to be not effective for communication. Sometimes dealing with notations of this kind for the first time requires learning (it is like learning a new language).

Tabular Tables can be used to express some models. They are especially very flexible in making elements of the model better classified (using columns). They employ textual notation to represent the contents of their cells. They are very useful in representing dual relationships using cells intersections but they will pose some problems when the number of relationships increases.

Graphical (diagrams) Relies heavily on our visual perceptions first and our interpretation skills second. They are very powerful in conveying information far easier than textual and tabular data, although they usually employ rely on text in most parts and rarely use tables. The power of graphical notations comes from the different variables they can employ. Those variables are known in graphic design as Bertin's *Semiology of Graphics* from his book *"Sémiologie graphique: Les diagrammes - Les réseaux - Les cartes"*. Bertin defined the following eight variables: horizontal position, vertical position, shape, value, color, orientation, size and texture.

6.1.2 Graphical Notation

6.1.2.1 Introduction

Graphical notations or diagrams are considered to be among the best efficient notations to express most models, especially descriptive ones. Among the most famous notations in our days we can cite the Unified Modeling Language (UML) which defines 13 types of models, all of which use Graphical Notation. The primary reason behind adopting graphical notations in most models is *Communication*. Diagrams are able to convey information contained in models better than other notations such as text or tables. With graphical elements, we can share easily the captured data inside a model with different people from different backgrounds and experience-levels. We repeat again, as mentioned in section 1.1 on page 7, that *communication* is a critical success-factor in software development to underline the importance of designing carefully notations and languages.

6.1.2.2 Principles for an effective graphical notation

We will present in this subsection a list of principles that are known to provide better effective diagrams when applied. This list relies heavily on the list and research performed in [Moody 2006] and presents a quick overview of each principle.

Discriminability Refers to the ease of differentiating diagram elements. There are two types of discriminability: absolute (differentiate elements from the background) and relative (differentiate between different element types).

Modularity Decompose complex models into smaller modules that are perceptually and cognitively manageable.

Emphasis Emphasize on important elements of the model and allow filtering of second-class elements.

Cognitive Integration When using multiple diagrams in modeling, they need to be easy to navigate in. The user should be able to have an integrated mental representation of multiple diagrams if any.

Perceptual Directness Employ direct representations which do not require cognitive efforts in order to be understood. This is can be done by making the representation share some important properties with what it represents.

Structure The way we group diagram elements.

Identification Refers to identifying the correspondence between diagram elements and the represented world, or in some cases the correspondence between these elements and the graphical convention in use.

Visual Expressiveness Refers to the number of variables used to encode the diagram (basically Betrin's variables).

Graphic Simplicity This is related to the number of graphical conventions used in the diagram. Usually, it corresponds to the number of employed symbols in the model.

6.2 Task Models notations

In this part we will present a review on different notations in Task Modeling. The work that led to this part was done with the help of three HCI students from the University of Toulouse. They worked with us particularly on the notation and interaction aspects of Task Modeling.

6.2.1 Representing the structure of a Task Model

Since the birth of Task Analysis and later Task Modeling, different notation languages were use to represent the structure of the captured models. These notations are found in different forms: text, tabular and graphical. In this section we will give a review of major existing notations and will focus more on graphical ones. The most popular form of representing hierarchies is trees which were used first by the HTA model. In this regard, the notation work will not involve how to structure the model but how to represent its constituents ; mainly different nodes. Thus, finding a way to represent conceptual relationships does not require a long discussion. On the other hand deciding how to represent communicative relationships and external elements pose a challenge in task models.

We will start first by reviewing the different representation methods for communicative relationships (basically task flow operators). Next we will discuss how we could integrate foreign elements to the task core structure (mainly objects) inside the model.

6.2.2 Hierarchical representation

6.2.2.1 Task level flows

In this kind of notations, task flows description are represented by additional graphical elements to be found at the parent Task Level. The presence of this element indicates that it should be applied to all direct subtasks of the parent task. Those notations are compatible with task models which attaches the task flow operator to each task. The actual details can vary considerably. For instance in HTA, task flows are considered as plans. Each plan is attached to its correspondent node. the plan contains a procedural-like code which describes how subtasks should proceed to execute (see figure 6.1 on the next page which shows a classical example of HTA).

The second type of notations takes advantage of the more advanced semantics that the core model provides. These notations rely on models describing a set of well defined task flow modifiers (mainly operators). This precision allows the notation to employ simpler terms instead of verbose procedural-like scripts. Usually the notation will attach the operator's name or symbol to the target node. We can distinguish two types of attachment in this regard: internal and external.

Internal Operators are drawn as part of the task node. They are more like an attribute than a modifier for subtasks or part of the hierarchy. This type of notation has the advantage of compressing the model by providing more space and it generally produces pure tasks hierarchies (as from the reader's perspective the model contains only a tree of tasks). However, when reading the model, one needs to look inside the node and localize the part where the operator indicator is drawn. This has a serious concern because task flows are important elements of the Task Model and need to be more visible and explicit in their representation. For an example of this notation you can refer to figure 3.7 on page 45 based on Amboss.

External Operators are drawn out of the task node itself but they remain attached to it using a link. In this type of notation, the operator has a

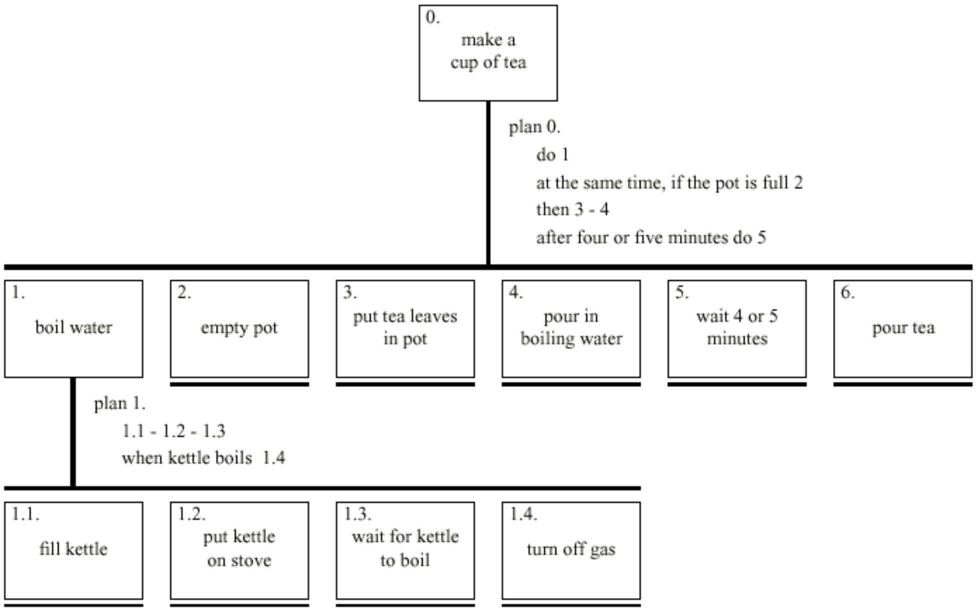


Figure 6.1: *Make a cup of tea* Task in HTA

more explicit presence and exists between the parent node and its subtasks making it clearly seen as some sort of relationship. Using this notation method, identifying different task flows becomes easier as the operators in this case are represented at the same level as their respective tasks. But it contributes to complicating the hierarchy structure by adding an additional notational level with every new task, and by making the hierarchy heterogeneous (tasks and operators are at the same level giving the illusion for some that this represents a tree of tasks and operators). For an example of this notation you can refer to figure 3.2 on page 36 based on K-MADE.

In summary, operators are built-in the task node this kind of notations. In other words we attach the operator to the node allowing the presentation to show that this operator applies to all subtasks (or to the parent task if it is unary).

6.2.2.2 Horizontal Flows

Another method to represent task flow operators is making them occupy the horizontal space of our model. This notation is very close to how we write expressions in arithmetics. Tasks plays the role of the operands and task flow descriptions play the role of the operator. This notation fashion has the advantage of both making the operators more explicit in the representation and preserving at the same time the hierarchy coherency and size. This notation goes further in compressing the hierarchy using its ability to express complex flows without requiring superfluous levels. However, it has some readability problems. Mainly, it requires either learning or an additional notation to help prioritize operators. The most famous example that uses this notation is the CTT model. Figure 3.5 on page 42 gives an example of notation in CTT.

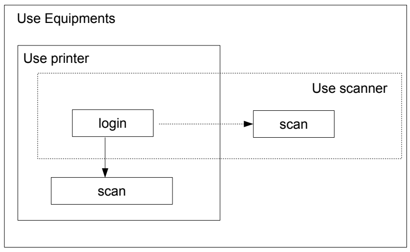


Figure 6.2: Sample heterarchy using Venn diagram

6.2.3 Heterarchical representation

6.2.3.1 How to represent a heterarchy

Heterarchies are a more general form of hierarchies that allows its nodes to have more than conceptual link. In our case, it allows the subtask to belong to more than one parent-task. This special property can be of a very valuable importance to create more valid models thanks to its ability to capture very complex real-world situations which their components are not forcibly organized or structured but inter-connected. Most scholars agree about the expressiveness and fidelity power of heterarchies but providing the right notation remains a problem. In this section we will take a look at some methods that can help us represent heterarchies.

Tree with replication The first notation solution uses traditional tree employed in hierarchies with an additional feature: node replication. The idea is to replicate any node as necessary to associate it with its parent. This notation is simple to implement and takes advantage of the existing familiarity of users with hierarchies. However, it makes our tree more complex if not impossible to read. Not to forget the confusion that can result from drawing the same node at multiple places inside the same model.

Tabular This is not a graphical notation, it uses tables to model heterarchies. This solution seems interesting at first but its problem becomes apparent when we wanted to represent a model with multiple levels, as for each level we will need a new sub-table.

Venn diagrams Venn diagrams are well used especially in the Set Theory. Their ability to represent sets gives us a clue how to employ them to represent heterarchies. The idea is to formalize heterarchies into sets and then map them into a Venn model. Graphically speaking, the solution might be easier to understand as figure 6.2 shows. We have a task named *login* linked to the task of using two distinct equipments. The notation clearly shows that this task belongs to the *Use Printer* and *Use Scanner* activities at the same time.

6.2.3.2 Task Flows

Representing task flows into hierarchies depends largely on the adopted notation (see above for examples). In trees, the same options that are available to hierarchies can be adopted. In tabular notation the challenge is greater, this is mainly due to its poor expressive power compared to graphical ones. The solution in this case it is basically using special textual syntax, special flags or colors. We will detail further representing task flows inside Venn diagram because it has some interesting properties compared to others. The following discussion talks about task flows in Venn diagrams in general and it does not limit itself to the case where they are used to represent heterarchies; as we will see Venn diagrams have some advantages even for hierarchies.

Contrary to trees which are links-oriented, Venn diagrams are set-driven from a mathematical point of view, which could be translated to a containment-based from the notation perspective. This is means the use of containment instead of links to represent membership. In our case, we can use this representation style to model our conceptual relationship "subtask" by drawing the subtask inside the parent task where it belongs. This choice leaves lines and arrows to be used exclusively by communicative relationships. Generally, employing arrows can spare us from using labels or new symbols and thus provide cleaner diagrams. In addition, arrows and lines have a strong history of usage to model flows. Another advantage we can cite is making it possible to have a task flow from two tasks of different decomposition levels.

6.3 Hamsters Notation

6.3.1 Diagram Structure and Tasks

In its conceptual form, our model is structured as a hierarchy to provide valid models having high level of fidelity to what they are modeling. However, our notation uses a tree graphical form to represent tasks which is essentially used for hierarchy-structured models. We highlighted above that heterarchies are challenging to represent in a notational language. May be the best choice would be using Venn diagrams but after experimenting with it we identified a major problem related to this kind of representation which is *space*. Venn diagrams consume much space making it very complex to represent deep level of details. The second problem lies in a decrease in the level of readability. Even if we consider that the space is unlimited, we will confronted with major complications related to diagram navigability. Further, as the number of decomposition levels increases as the model becomes very complicated. It will present a first challenge to approach graphically two tasks sharing the same actions. In addition, the intersections caused by this notation can be confusing as the number of overlapping lines keeps growing. At the end of our thesis, in the prospects section on page 117, we will try to give some possible solutions to the navigability problem in such diagrams. Note that any possible future solution can be attached to our model without modifying its core as in its pure form is structured as an heterarchy.

The most important element of any task model is *Task*. In our notation, the diagram is basically a hierarchy or a tree. The nodes of this tree are the tasks. We used a combination of various perceptual dimensions to represent a task element (see figure 6.3 on the next page; operator was omitted because it will be discussed in the following section). The dimensions used to encode a Task node graphically are:

Vertical position It identifies to which level of hierarchy this task belongs. This corresponds to the decomposition level. The order of the hierarchy is not specified explicitly to allow more freedom but usually it is top-bottom.

Horizontal position It encodes the order in which tasks are executed (starting from the left).

Shape We use a rounded rectangle as a container shape to symbolize a task. The perimeter is by default a continuous line to denote a mandatory task. Optional tasks are drawn using a dotted line. When representing an *OperatorTask* this shape and all its contained children are ignored.

In addition to this basic representation, the task node contains additional graphical elements that are encapsulated inside the shape container limits. Those elements in order of representation from top to bottom are:

Icon It identifies the type of the Task. Each task type defined by our model has its own icon. Our selection of icons were carefully chosen and evaluated so they reduce the required cognitive overhead. The table 6.2 on page 96 lists all of our task icons with their respective Task elements followed by the reason behind their design.

Label This is a simple text fields containing the task name as given by the analyst. It has a special *value* perceptual dimension because it uses a bold character to make it more explicit. Users usually look for a specific task based on its name.

Features This a horizontal container used to show various features that are present in this task. More practically, it can be used to indicate if the task has a Guard or an Iteration property. This container is extensible and can be used to add system-specific features.

Additional containers The task node can have additional containers or compartments. This feature can be customized allowing the analyst to add additional details to his/her model. For example he can add a compartment for to show the list of pre-conditions (see figure 6.3 on the next page; it has a compartment named “conditions”).

Operator The attached operator to this task. The next section will detail the used notation for operators.

The *color* perceptive dimension in Hamsters notation is used in different places. This concerns graphical elements including icons. To have a regular *default* color scheme, we identified a set of colors and associated each one with a set of relevant properties (see table 6.1 on the facing page). These basic colors were than harmonized using the Kuler tool from Adobe [1].

[1] http://kuler.adobe.com

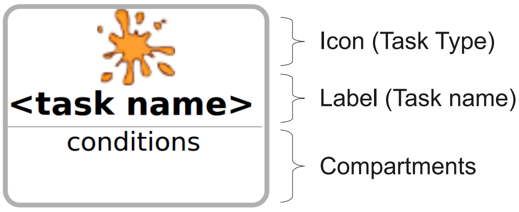


Figure 6.3: Task Element notation

Property	Color
Technology	Blue
Cerebral	Violet
Communication	Brown (humanity)
Reassuring, Comfort	Green
Warm	Orange
Solid, reliable	Metallic (sparkling)
Professional	Gray

Table 6.1: Colors properties from a western perspective

As mentioned above, this is a default scheme because color signification depends on a very complex factor which is the cultural background of the user. Our table builds on common colors meaning found in the West. A customization is possible to adapt our notation to a different culture but this can lead us to multiple notation problems. When exchanging models in their native format, the model will always show up using the analyst's own configuration of notation. However, in other formats like print, this can be confusing. We think that analysts who are spread over different cultures should use one agreed-upon notation to avoid confusion and communication problems.

6.3.2 Operators

The second most important elements to represent in our diagram after Tasks are *Operators*. In our model, each task can have its own Operator. OperatorTask relies exclusively on the Operator's notation inside the diagram as it does not have a *shape* perceptual dimension *per se*. In our notation we chose the following perceptual dimensions to encode an Operator graphically:

Vertical position It indicates the level of this operator inside the hierarchy. In addition, it helps the analyst determine the order of evaluation for complex task flows.

Horizontal position It Specifies the order in which operators should be executed by their parent operator.

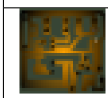
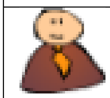
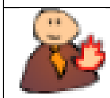
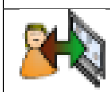
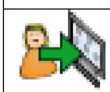
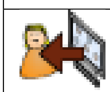
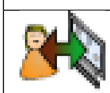
Icon	Task Element	Description
	AbstractTask	A spot of paint. Represent an undefined form referring to *abstract* art.
	SystemTask	Printed Circuit Board. The darkness makes it seem more like a black box.
	UserTask	A basic *user* icon; will form the base for sub-types of UserTask.
	CognitiveTask	A partial user icon (the head) with a balloon inspired from comics indicating the presence of a *cognitive* process.
	MotorTask	The basic user icon with an emphasis to show his hand referring to task which requires *motor* skills.
	PerceptiveTask	The basic user icon with an emphasis on *perceptual* senses.
	InteractiveTask	Represents a user facing a computer screen, implying some kind of interaction.
	InputTask	Basic icon of InteractiveTask with an arrow from the user to the screen implying an *input* action.
	OutputTask	Basic icon of InteractiveTask with an arrow from the screen to the user implying an *output* action.
	InputOutputTask	Basic icon of InteractiveTask with a bidirectional arrow implying an *input/output* action.

Table 6.2: Icons of Task Elements

Shape We use a circle to symbolize an operator.

Inside the circle, a textual symbol indicates the operator type. We chose to adapt the same symbols used by the CTT task model to take advantage of their relatively wide usage in the community. For a complete list of these operators see table 3.1 on page 39.

6.3.3 Objects and Information Flow

Most task models take into consideration the presence of objects in their meta-models to varying degrees. When it comes to notation, these objects do not have a direct representation inside the model. Usually they are added to the model using dialog forms and dot not show up in the diagram. In our model, we wanted to allow objects to have presence in our diagram. Objects in Hamsters are represented using two perceptual dimensions (see figure 6.6 on page 100 for the an example graphical representation):

Shape Objects are represented using a rectangular shape. The shape was inspired by the rectangle used in UML diagram classes helping users having some familiarity with this type of diagrams. This shape serves as a container for some additional sub-elements:

Label The object name inside the model.

Attributes compartment It serves as a container for this object's attributes. Attributes are represented as a set of vertically listed labels with each label referring to an attribute name.

Color We preferred to add the color perceptual dimension because of the strong similarities between the shape used for Task and the one for Object (rounded rectangle vs. rectangle) which can cause confusion. Adding a special background to identify objects make it easier for users to filter them out of tasks.

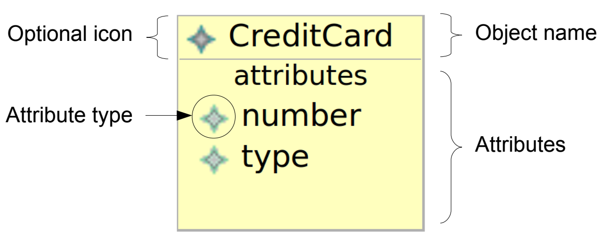


Figure 6.4: Object notation

The major challenge that results from this choice is a considerable increase in the diagram elements condensation: making it very difficult to read (see figure 6.5 on the next page for an example). Adding objects to the model notation means more space consumption but more importantly means defining their location. Our model is represented as a hierarchy so having external elements to this hierarchy pushes us to find a way to put them inside this hierarchy without interfering with its core structure. The proposed solution to this problem is making objects visibility optional and second developing a special interaction that shows objects only when they are relevant. For instance, we can develop the following interaction techniques in order to deal with condense models:

- Objects will be shown only when tasks manipulating them are selected.
- Show only objects belonging to the abstraction level the user is working at.
- Show objects only when the communication flow they are attached to is selected.

Unfortunately, those example solutions are interaction-based and can be only implemented in the editor or viewer. They do not solve the problem when printing the whole diagram for instance. Nevertheless, we believe having it possible to represent objects inside the model is better off than disabling completely this feature.

Information exchange in Hamsters is represented using directed arrows. Objects, than, can be attached to these arrows indicating that they are sent along this communication flow. In their default configuration, communication flows are used to link two tasks directly. Additionally, to support Hamsters concept of *ports* (see 5.3.7 on page 80), communication flows can link two tasks indirectly through their ports. Figure 6.6 on page 100 gives an example of both types of communication: (a) direct communication flow, (b) communication flow using ports.

6.3.4 Hamsters notation reviewed

In this section we will review our notation by projecting different choices to the principles listed in 6.1.2.2 on page 89:

Discriminability Hamsters employ two techniques to achieve absolute discriminability. The first is forcing a minimum size for all elements (primarily implied by their mandatory content; e.g. icons for tasks). The second is contrast, Hamsters icons, lines and labels are designed to have a sufficient contrast with the background (which is white). For relative discriminability, Hamsters employs various perceptual dimensions to encode different elements.

Modularity Hamsters supports diagram partitioning allowing the user to divide the model into smaller chunks. Those chunks can be later assembled using Task references in a higher level model.

Emphasis Special emphasis consideration were given to tasks compared to other graphical elements. Communication flows were given additional emphasis compared to hierarchy links. The reason is that the latter links can

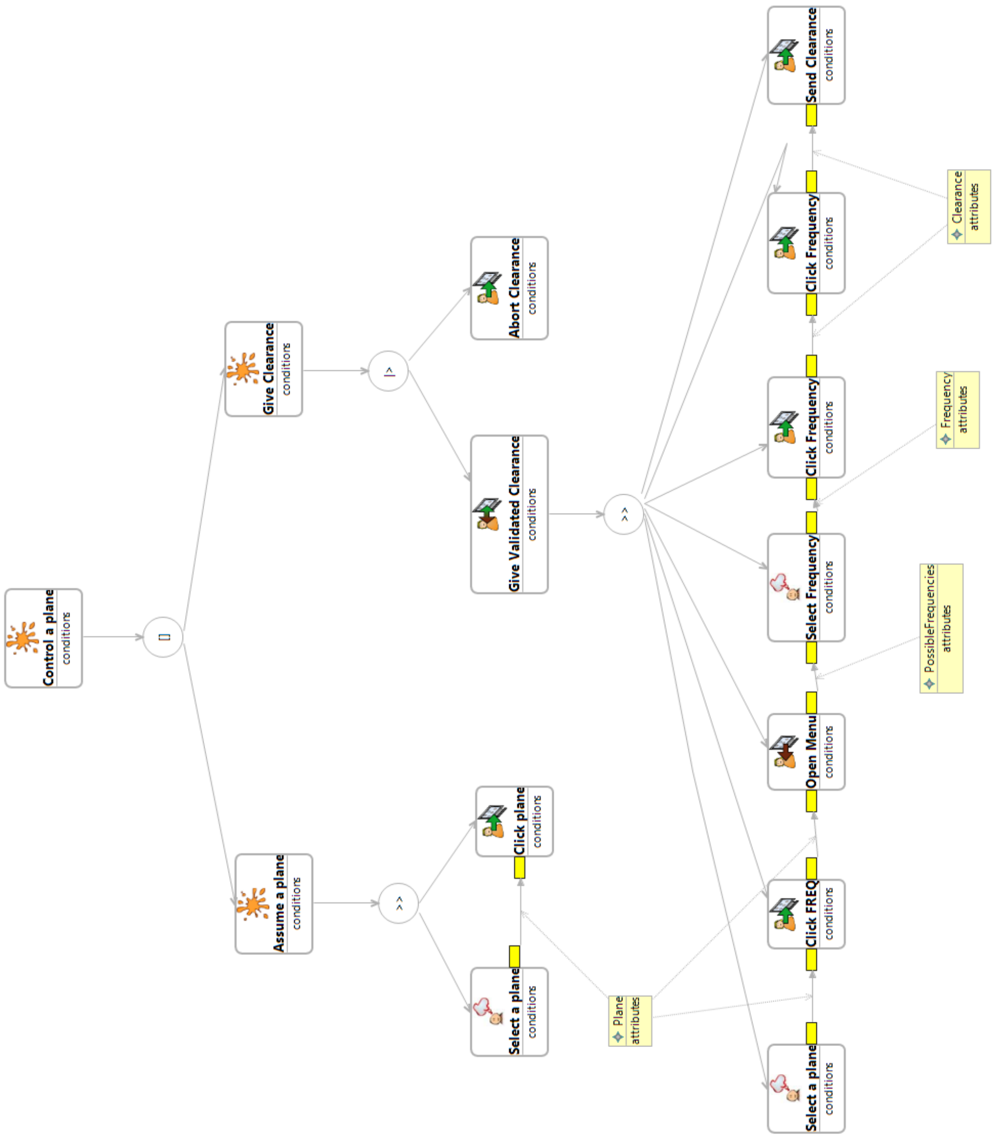
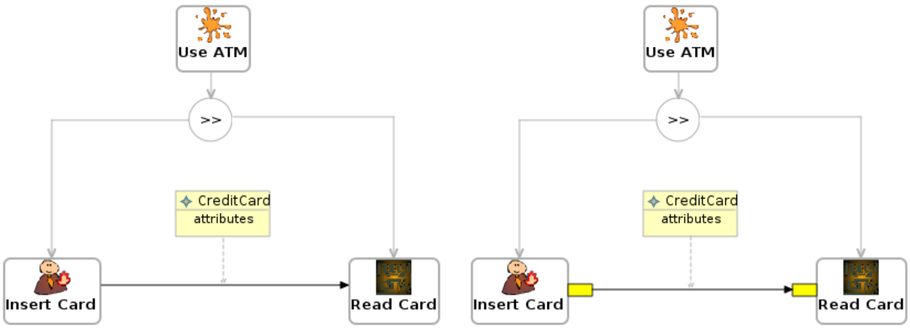


Figure 6.5: ATC example model in Hamsters



(a) Direct communication flow (b) Communication flow using ports

Figure 6.6: Communication flow notation

be identified easily thanks to the additional perceptual dimensions (vertical positions of elements and orientation). Inside tasks, a special emphasis was given to the task name which plays the role of its identifier inside the diagram making it easier to locate tasks.

Cognitive Integration Hamsters defines only one diagram to define all task model aspects preserving cognitive integration through all models. In addition, Hamsters diagram partitioning makes it efficient to navigate between different diagrams in an integrated fashion.

Perceptual Directness All icons designed for Hamsters are symbols which perceptually resemble the objects they represent. This helps people to infer their meaning without explanation or reference to a legend. The symbols used for operators do not have directly perceptual because they are representing abstract notions and are based on existing notations which could make adopting our notation easier. When it comes to relationships, we use vertical directed arrows for subtask relationship which can be easily perceived as a decomposition concept. Communication flows use directed horizontal arrows with attached objects giving the illusion of object transfer.

Structure We use a well defined structure in our notation which is *Hierarchy*.

Identification Our diagram is identified by its root task. Users are encouraged to place this node at the top of the diagram.

Visual Expressiveness We tried to introduce different perceptual dimensions as possible in order to encode as much data without flooding the diagram with superfluous flags and indicators. The resulting language can be used to express almost all the model semantics.

Graphical Simplicity Our notation defines three major graphical elements: tasks, object and links. This leaves us with only 3 categories clearly below the *span of absolute judgment* which is around 6 categories [Miller 1956].

Chapter 7

Hamsters Implementation

7.1 Hamsters CASE tool

7.1.1 Hamsters CASE classes

In software engineering, CASE (Computer Assisted Software Engineering) is the field responsible for developing and creating support tools that can help developers during software lifecycle. From the course of *Advanced Cooperative Systems* (prof. Englebert), we have learned that CASE tools can be classified along two axis. The first is about their presence along the lifecycle, they can be categorized as *horizontal* meaning they provide support for different cycles, or they can be *vertical* and thus are only able to support one axes. The second axis is their contribution level. Some CASE tools are meant to be used at *upper* levels: analysis process, requirements, specification... Others are more oriented toward lower level processes: implementation process, editors, version control, debuggers... When used at lower levels, the tool needs to produce high-quality, error-free artifacts.

The implementation of Hamsters will result in a CASE tool that will contribute to simplify engineering of software systems, mainly targeting interactive ones. As a task model, Hamsters could be classified more as an *upper-vertical* CASE tool. This classification is not final but concerns the core features provided by Hamsters in its most primitive form. Hamsters can be extended in various ways allowing it to extend in time (making it *horizontal*), and to approach implementation (making it *lower*).

It is an upper level tool because its main function is modeling Task Analysis which is a higher level activity that can help us understand the system, elicit requirements, and evaluate system performance and design choices. Nevertheless, it should be noted that Hamsters can be extended to provide low level features. These features concern mainly generating documentation and training artifacts, or to provide output that can be used by other lower-level CASE tools to produce interactive systems. This is mainly done by mixing what we captured into the Task Model with the System Model. Finally, Hamsters is mainly vertical because it is used in some well defined processes of the software lifecycle. This limitation does not mean that any model or tools (evaluation, simulation) produced by Hamsters cannot be used in different processes. Further, Hamsters depending on its extensions and context-of-use can be used for various goals and thus can extend in time to support multiple cycles. We already showed using the goal model in figure 2.1 on page 31 the wide variety of uses for a Task Model that can span different software processes.

7.1.2 Hamsters CASE architecture

In this section we will provide the architecture of Hamsters through its different modules. This list is partially based on the one we saw in *Advanced Cooperative Systems* course.

7.1.2.1 Model Data Structure

This module is responsible for holding the model data in memory. It maps our model elements into real objects so they can be processed easily by the different other modules. In its implementation, this module defines a class for each classifier in our meta-model. Each classifier will have its own attributes and methods. References on the other hand are transformed into local variables. Those variables can be mono-valued in the case of a 0..1 or 1..1 multiplicities, or multi-valued (collection) when having a multiplicity greater than 1 (e.g. 0..2, 0..*). In our Java implementation, all of these classes are accessible only via a set of interfaces. The advantage of this choice is total abstraction from the implementation details. Any external module to Hamsters should only access the model using these interfaces.

7.1.2.2 Graphics and Interaction

Hamsters features another module providing different services for graphics and interaction. The graphics sub-module allows Hamsters to draw its element into the canvas (The graphical component on which we will draw our model; it is usually in the form of a rectangular white container). The module relies on a lower-level Model Data Structure that in addition to holding model elements, it contains additional graphical properties such as the position, size, color... The Interaction module on the other hand provides users with the necessary tools to interact with the model through its representation. This module plays the role of the Controller in this Model-View-Controller architecture (the Model Data Structure being the model and the Graphics module being the View).

7.1.2.3 Model Checking and Processing

Hamsters in its conceptual model has a strict formal definition. However in practice we cannot force this strictness on users from the beginning. It is like forcing a Java programmer to write a whole Java class without generating any syntax error which is impossible. Hamsters takes into account the temporal structure of human cognitive activities such as exploration, understanding, communication and design. It functions by default in a *Tolerant Mode* state. While in this mode, users have much freedom in the way they build their model. They can insert unlimited number of isolated tasks and place them wherever they want, they can create objects that are not connected to any communication flow... This way the user can construct his/her model in parallel with the exploration phases for example. Finally, when the user finishes his/her model, he or she can run the model checking module which will in turn process the model and identify any anomalies or errors. The table 7.1 on the following page provides the list of major example checks performed by Hamsters model checker.

7.1.2.4 Intelligence module

Intelligence modules are usually added to CASE tools to provide automated features that require some intelligence. Hamsters has one predefined intelligence module which is simulation. It allows analysts to run different scenarios following the model description they gave. The module will execute the tasks according to their flows, will interact with the analyst when necessary (to make non-deterministic decisions for instance) and will monitor objects states along the simulation lifecycle. The simulation module can detect exceptional flows and highlights them too.

Hamsters can be extended to include further intelligence modules, for example we can develop:

Cognitive Validator It can measure the cognitive load of various tasks and notify the analyst if there is an overload somewhere. Those extensions depend primarily on the purposes we defined to use Task Analysis and Modeling.

Documentation Generator It can generate full system documentation based on task descriptions.

Exporter It can be used to export the model into various formats. For instance export a scenario into a Flow Diagram, or generate a basic UML use-cases.

7.1.2.5 Model store

This module provides services to make the Task Model persistent . During the first phases of Hamsters, we wanted to use a standard persistence representation for our model. The OMG XMI format was chosen during a meeting on technological choices, mainly for its relative wide adoption by various CASE tools, and second it will make it easier for us to employ existing tool that can read an manipulate the XMI format. For example to run model transformations from and to our model format; we use existing transformation tools to export a model defined in Hamsters into another tool format (CTTe for example).

7.2 Meta-CASE

Before proceeding into the development of Hamsters support tool, we discussed various choices. Actually, those choices are coupled to the technology options we have in hand to develop our tool. We had three paths, either to develop everything from scratch (i.e. develop all the aforementioned modules from scratch), use some external libraries and modules, or use a Meta-CASE tool.

The first option offers a fully controlled solution with complete freedom on how to implement the final application. However developing everything from scratch is time consuming and can be counter-productive: instead of focusing on how to implement our findings, we will be likely dealing with other complexities such as lower-level graphics.

Error (e) /Warning (w)	Reason and Solution
Orphan Task (e)	A Task element found but it does not have any parent task. The solution would be integrating this orphan task into the hierarchy by making it a subtask of an existing Task.
Only one hierarchy is allowed (e)	The model checker find two distinct task hierarchies inside the model. To solve the problem the analyst should either relate those separate hierarchies to form one hierarchy, or create a new model where he or she can put one of the hierarchies.
Leaf OperatorTask not allowed (e)	An OperatorTask element was found but without any subtask. The analyst should append valid Tasks to this OperatorTask or remove it from the hierarchy.
Operator expects at least n operands (e)	The operator attached to the task requires at least n operands but there is less than n subtasks. The analyst should add enough subtasks required by the parent task operator.
Invalid reference (e)	A TaskReference cannot find the Task it is referencing. The analyst should verify that the actual task exists. This usually results when the proxy cannot resolve the actual task (not able to access the referenced model file for instance).
One subtask (w)	This warning indicates that there is a task with a lonely child.
Abstract Task is leaf (w)	This warning indicates that there is a an abstract task that does not have children tasks. This type of warnings helps analysts avoid vague task definitions and push them to give more concrete details.
Empty communication flow (w)	A communication flow should have at least one object to transport.

Table 7.1: Example of errors and warning checked by Hamsters

The second option still provides us with a high level of freedom while allowing us to use some existing libraries to support the development of some modules. The most important module that concerned us was the Graphics and Interaction module being almost the most complex to develop. For this we started looking for some third-party libraries that are conceived to create diagram based tools. This search was limited to Java-only libraries (the chosen programming language of the implementation) and we found a library called JGraph. This library which started as an Open Source project, allows the developer to create graphs using a swing-like components architecture. This path was tempting but it turns out that some additional features that were needed by our application were only available with a commercial license.

The third option was to use a Meta-CASE tool. A meta-CASE tool is a software that helps in the design and generation of CASE tools. Our choice finally was to use this option thanks to the different services and tools it provides that can accelerate our development. Being our technology of choice, we will devote this section to it.

7.2.1 Concept of meta-CASE

The idea behind this concept is the strong similarity between different graphical CASE tools, they share almost the same modules. Thus, a meta-CASE tool provides a set of generic components that can be customized as needed to generate finally a CASE tool compatible with our specification. Furthermore, the generated tools are the result of rigorous and continuous development carried by expert people from different backgrounds: computer graphics (visualization and editing), modeling and meta-modeling, etc. Another important advantage is the flexibility of meta-CASE tools and how they are developed to embrace change and evolution. In traditional approaches, an error in the meta-model can cost the implementation a lot (requiring sometimes a whole new restructuring). In meta-CASE tools, we worry only about the model description at a very high level independent of most implementation details enabling us to adapt our tools to change in the method, the model, the notation etc.

As a consequence, Meta-CASE tools are usually available in the format of a very specialized CASE software that enables users to provide a higher level descriptions of the required tools. Based on these *meta* descriptions, the tool will alter the generic content of the components it supports. This flexibility varies according to the selected meta-environment. Different types of meta-CASE tools exist, some are able to support almost all functionalities required by a rich modern CASE tool, some are limited to a subset of features provided by a CASE. Our tool, Hamsters, is basically a model-driven CASE tool; meaning that we will not need advanced features such as providing our own methodology and process support. Therefore, next sections will focus on meta-CASE tools that are model-driven.

7.2.2 How it works?

As mentioned above, meta-CASE tools relies on providing some generic components that can be customized later to produce a more complete and more specific CASE tool. Those components are almost the same as provided in section 7.1.2 which lists the major required modules in Hamsters architecture.

These components are usually present in every major meta-CASE tool, but the difference lies on how they generate and integrate them into the final product. Basically, we can identify two approaches in generating CASE tools:

7.2.2.1 Compilation

Following this approach, the meta-CASE generates a set of complete code files (e.g. java classes). After a successful generation, the tool will proceed by compiling the generated source code which will result in a new compiled dedicated software. This solution is considered very efficient because the generated code is compiled and optimized to support the model we defined. Even at the implementation level, this approach is not very difficult to implement. The tool can rely on code-templates, providing generic source code of components, than filling in the gaps to insert model specific attributes and code. However this solution introduces some problems because of the conflicts that can be produced between the fully-generated code and user code; making its maintainability a challenge (the solution would be providing support for intelligent synchronization).

7.2.2.2 Interpretation

The meta-CASE plays the role of an interpreter and usually runs the generated CASE tool inside its environment. This solution does provide a slower tool because the interpreter needs to load and parse the meta-data each time. It is also more difficult to implement, requiring advanced generic interpreters and the ability to integrate generated tools inside the same environment. On the positive side, it is the most flexible. It is easier to alter our CASE tool in this approach without recompiling the tool every time we change our model definition as compared to the first approach. Another interesting feature is that the same environment can be used to edit various models at the same time, reducing the number of CASE tools and providing a unified software environment (same UI, same notation basics. . .). Moreover, having interpreters makes it easier to communicate and exchange models thanks to their structural and representation similarities.

7.2.3 Architecture

CASE tools are usually coded into a two level architecture. The first level defines how the model is structured, its notational elements, . . . The second level is the model themselves. In Meta-CASE tools there is an additional higher level. This level or layer was introduced in order to make the second layer (which corresponds to CASE first level) more flexible. In fact when applying modeling terminology, this level would correspond to the Meta-Meta-Model, the second to the Meta-Model and the third to the Model. The power of meta-CASE tools is their ability to understand Meta-Models described using the Meta-Meta-Model they define. Based on this description, it will generate the required modules. The figure 7.1 on the next page, based on one from a white paper by MetaCase Inc. [2004], gives an overview of supported levels in both meta-CASE and CASE tools. The figure 7.1a shows how a CASE tool is usually composed of two levels. The first one defines how different components should be developed, the problem here is that all these descriptions are hard-coded directly

(a) CASE architecture levels (b) Meta-CASE architecture levels

Figure 7.1: CASE vs Meta-CASE architecture

into the code. This architecture will require code modifications whenever the model evolves or changes. The second figure 7.1b shows how a Meta-CASE tool differs in architecture by adding a new higher level. This level defines the language in which the CASE developer should describe his model, notation, store... Its power lies in the ability to read any description and then generate the respective implementation. Most modern implementations of meta-CASE rely on the Meta-Object Facility (MOF) to describe their meta-models. MOF is an OMG standard that has its origins in UML meta-models definition. In OMG standards, the meta-modeling architecture consists of four layers (see figure 7.2 on the following page for an example based on the UML class diagram). The first three layers correspond to the generic three layers we presented in figure 7.1b for meta-CASE tools. MOF is concerned with the definition of the language to use at the M3 level. It is a strict specification of meta-modeling which allows designers to describe the structure of their meta-models in a formal way. In essence, it is like an abstract syntax for languages. Moreover a direct analogy is usually drawn between EBNF (which used to describe programming languages) and the MOF which is used to describe meta-models. The last layer M0 in the OMG meta-modeling architecture is said to represent real world objects. In the case of Task Modeling, the M0 level would correspond to tasks instances ran inside scenarios (usually created by users or more easily using the simulation module).

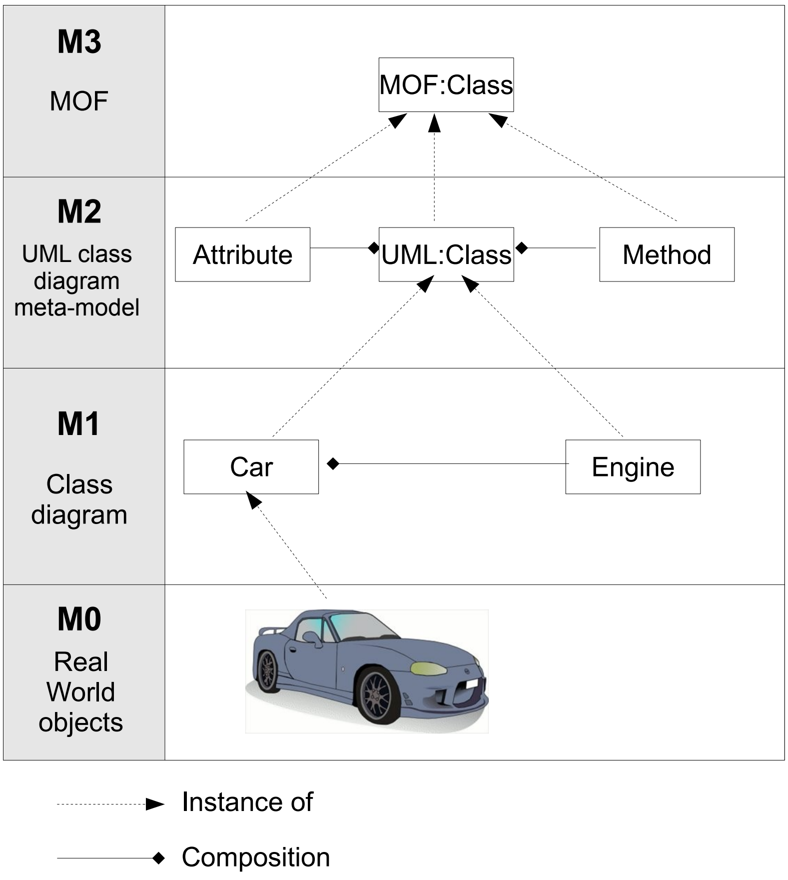


Figure 7.2: UML class diagram MOF architecture

7.3 Hamsters Design and Implementation

7.3.1 Eclipse Modeling Project (EMP)

After discussing the various implantation options we had and detailing our method of choice which is using meta-CASE tools, we will now present more practical and concrete information about the meta-CASE technology we have chosen.

The meta-modeling technology we chose to develop Hamsters is the based on a collection of frameworks from the Eclipse Modeling Project. According to the Eclipse website: "*The Eclipse Modeling Project focuses on the evolution and promotion of model-based development technologies within the Eclipse community by providing a unified set of modeling frameworks, tooling, and standards implementations.*" We selected the EMP for our implementation for various reasons:

1. It is free and Open Source.
2. It builds on the Eclipse Platform which demonstrated itself as a very reliable platform.
3. To take advantage of the Eclipse Plugin Architecture.
4. To make Hamsters run on multiple platforms and operating systems.
5. EMP in most of its parts is based on open standards related to meta-modeling, mainly the OMG specifications. For instance EMP implements a functional subset of the MOF standard.
6. To take advantage of model transformation technologies built for it.

After giving a short introduction to the Eclipse Modeling Project, we will detail the different frameworks that constitute the building blocks of EMP.

7.3.1.1 Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework is the basic framework used to describe meta-models (the abstract syntax of our models). It corresponds to the M3 level in the OMG meta-modeling architecture. In fact, EMF is a partial implementation of the MOF standards. Starting from a meta-model description, EMF is able to generate the necessary code (java classes) which provides services to access and edit the model. EMF is responsible also for providing the storage module for the model.

7.3.1.2 Graphical Editing Framework (GEF)

The Eclipse Graphical Editing Framework was developed to provide advanced graphics features for the Eclipse Platform. The idea behind GEF is to take an existing application model (which can be described in any language not necessary using EMF) and create a rich graphical editor. More precisely the GEF framework provides the following sub-modules:

draw2D An SWT drawing library that provides a layout and graphic toolkit for drawing graphics.

Core GEF A set of GEF additional features that are commonly present in most graphical editors. It provides basic tools such as Selection, Creation, Marquee. It supports tool palettes, advanced editing system based on the *Command* design pattern. Finally and more importantly, GEF provides the necessary controllers that will make sure that our graphical representation is synchronized with our model.

7.3.1.3 Graphical Modeling Framework (GMF)

The Eclipse Graphical Modeling Framework was developed to create a bridge between the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). GMF provides a generative and a runtime components that enable us to integrate models described in EMF inside the GEF MVC architecture and finally produce a graphical editor. Usually after defining our meta-model in EMF (which corresponds to the abstract syntax definition), we proceed to describe the concrete syntax with the help of GMF by:

1. Writing the notation description. This is done by defining it inside the *GMF graphical file* which enables us to give a detailed description for each graphical element we will use in our notation.

2. Writing the tooling description. In this step we describe what are the tools that our final graphical editor should provide.

3. Linking the graphical description, tooling description and the EMF meta-model (Ecore model). All these relationships between these different three descriptions are written inside a mapping file.

4. Generate the basic code for our graphical model based on the information provided in the mapping file.

5. Customize the code and add additional features which are specific to our model.

7.3.1.4 Hamsters implementation using EMP

Using the meta-CASE tools provided by the Eclipse Modeling Project provided us with all the benefits mentioned in the meta-CASE section. It allow us to focus more on the application domain and not the application details. The development process using the EMP frameworks is very powerful and almost fully automated. However, it is not really straightforward as it may appear. In fact, after generating the code, we needed much time to quirk and fix it in order to be compatible with our specification. For instance to support scalable graphics (SVG), we needed to change some graphics classes to add the necessary code in order to draw the required graphical image. We were not able to annotate the custom code because we could not decouple it from the main drawing method which resides inside a java nested class. The solution we adopted in the end was to create some sort of patch files that update the generated code each time.

Another problem we faced is the forced Canvas element inside the graphical definition file. This element needs to be mapped to an existing model element which can cause some problems because it creates an additional container which does not exist really inside our meta-model definition. We solved this problem by mapping the Canvas element to our *TaskModel* element inside the meta-model. However, this choice left us with some bugs when we implemented diagram partitioning.

When adding elements inside the graphical editor, the mapping definition required from us to indicate into which container they should belong. The default behavior is to include them inside the TaskModel which corresponds to the Canvas (global graphical container). This default behavior created a problem for us as all tasks that the user will add will be considered as subtasks of Canvas and moreover this solution would be possible only if we consider the relationship between a TaskModel and a Task as a containment reference. To solve this problem, we used a GMF trick called *Phantom nodes*. Phantom nodes can be added to the Canvas without being forcibly contained inside any model element. The only problem with this approach is that all phantom nodes will be inaccessible by the root model element (which should be the rootTask of TaskModel) as they do not belong to any element inside it. To overcome this limitation, we needed to look inside the EMF and GMF internals to find a way to access Phantom nodes from the model without relying on any graphical or tooling component.

Finally, we would like to state that despite all these problems and its immaturity (especially GMF which is still considered in its early stages), EMP frameworks proved to be very usable and powerful enough to satisfy most of our requirements. More importantly their open architecture and extendibility allow us to implement any feature, all you need is finding the right extension location, integrating the feature and testing (re-running unit-tests is usually sufficient).

7.3.2 Hamsters Cognitive Dimensions

During the development of our tool, we wanted to take care of the some cognitive dimensions related mainly to interactivity. In this section we will give short introduction to cognitive dimensions than we will evaluate our tool according to these dimensions.

7.3.2.1 Cognitive Dimensions

Usability is an important factor in the success of most software products today and CASE tools are no exception. Unfortunately for a long time this concept was used in an informal way with various terminology and definitions clashes. As a result, it was always very difficult to design or evaluate systems usability in a systematic way. First attempts to solve this problem were relying on creating some sort of usability checklists [Nielsen and Molich 1990] or provide the designer with a procedural list of design activities he or she should follow [Wharton et al. 1994]. Unfortunately, those list based techniques were found to be not efficient in designing or evaluating even user interfaces [Winograd et al. 1996] let alone the overall usability. A better solution would be creating a common theoretical and craft-oriented ground to build on for usability engineers. This is the main idea of Green and Petre [1996] by defining a framework that can help design new systems or evaluate existing ones. This framework was introduced first in their paper “Usability analysis of visual programming environments: A ‘cognitive dimensions”’ and continued to evolve since then. The whole framework is based

Cognitive Dimension	Description
Viscosity	Known also as *resistance to change*. It is about how much effort is needed to make a change in the program.
Abstraction gradient	What is the flexibility of the notation in hiding/exposing details?
Consistency	As part of the interaction is learned, how much new ones can be guessed successfully?
Hidden dependencies	Are there dependencies between elements visible or hidden? How a change in one area reflects on the other?
Premature commitment	Is the program forcing the users to follow a forced method using strong constraints? Does the user need at times to take a decision without having all the necessary information?
Progressive evaluation	How the program is able to provide feedback on the current work without needing it to be in its final form.
Secondary notation	Does the program provide extra flags or notations that can carry informal additional information that the user needs to incorporate.

Table 7.2: List of Cognitive Dimensions

on Cognitive Dimensions which are a set of well defined design principles for user interface, user interaction and notation. Among the purposes and facilities that the Cognitive Dimensions framework provides we can cite [Blackwell and Green 2003]:

- To offer a comprehensible evaluation.
- To Use common terminology that can be comprehended by nonspecialists.
- To be not limited only to interactive systems but can be used for paper-based and other non-interactive systems.
- To be theoretically coherent.
- To differentiate between different types of user needs with high precision.

You can find a small descriptions of various known Cognitive Dimensions in table 7.2. Note that there are some common concepts between Cognitive Dimensions and Notation Principles (see 6.1.2.2 on page 89) that we omitted here and chose instead to concentrate more on interaction-oriented dimensions.

7.3.2.2 Cognitive Dimensions in Hamsters

After giving an overview on the Cognitive Dimensions framework and defining some major elements of it, we will try in this section to project these dimensions on our application's usability which you can find in table 7.3 on the next page.

7.3.3 Hamsters Application and Plugin

According to initial requirements, Hamsters should be able to integrate into the PetShop CASE tool. The goal was to link PetShop models with Ham-

Cognitive Dimension	Hamsters Evaluation
Viscosity	The user in hamsters deals basically with different model elements. The application does not provide complicated systems to support change but some basic ones. This includes the ability to apply a change in one shot to a whole selection of elements, though most of the time these elements need to be of the same type. The application provides unlimited undoes and redoes.
Abstraction gradient	In its default notation Hamsters does not represent the whole model content. However, it provides a powerful abstraction gradient by using compartments. Compartments are graphical containers that could be shown/hidden at request. Further when they are visible they can be collapsed or encapsulated leaving only their title visible. In addition, the property sheets can be extended to include additional attributes as needed.
Consistency	The user interface is based on the SWT toolkit providing a uniform interaction with basic UI elements. The notation relies on a common strict terminology, icons and cursors.
Hidden dependencies	Dependencies in Hamsters are essentially related to the model hierarchies (not to be confused with tasks hierarchies). Hamsters allow users to visualize dependencies using a TreeView components. In addition changes performed in one place are usually propagated to the others (thanks to the use of references to a one real central task).
Premature commitment	As explained in section 7.1.2.3 on page 102, Hamsters is executed in a *model-tolerant* mode which does not require that the model must be correct at any time. Users can build their models with a complete freedom without barriers neither worrying too much about constraints.
Progressive evaluation	The analyst can run the check modeler at any time to provide feedback about the model correctness. In addition, Hamsters relies on some tools provided by a project called Epsilon from Eclipse to provide direct graphical feedback inside the notation if the user enables this option.
Secondary notation	Hamsters allows the users to add personalize the notation with the aim to add additional informal infatuation to the model through two techniques. The first is using comment boxes which are graphical elements that can be connected to any element inside the model. The second is using appearance styles (colors, line styles. . .) and Hamsters layout-free notation.

Table 7.3: Cognitive Dimensions in Hamsters

sters task models, thus requiring Hamsters to expose its models to external tools. The idea is to create a bridge between Hamsters (Task Model) and PetShop (System Model) in order to support better the development of interactive-systems (mainly by using scenarios in both worlds modeled by Hamsters and PetShop). For more details about the idea of bridging the gap between Task Models and System Models and using them in a complementary fashion to develop interactive-systems see the work of Navarre et al. [2009]; Palanque and Bastide [1996]. At the same time, we wanted to develop a standalone version of Hamsters so Task Analysts can use it without needing another environment. The solution we proposed was to use a service-oriented architecture (SOA), exactly we opted for OSGi (formerly known as the Open Services Gateway initiative). OSGi a set of open standards of a java based service-oriented architecture released by the OSGi Alliance [2007].

To satisfy both requirements we opted to use the Eclipse Platform which has a built-in support for the OSGi architecture. In fact the Eclipse Framework uses the OSGi architecture for its rich and famous plugin architecture. Moreover, the Eclipse implementation of OSGi named *Equinox*[1] was adopted as the reference implementation by the OSGi alliance. This is an additional reason behind our choice to use the EMP in developing our implementation. More practically, using Eclipse we can deploy our implementation into either a standalone application (known as RCP: Rich Client Platform which runs on top of its own OSGi layer), or as a plugin installed on top of another OSGi layer. The first challenge we faced was that PetShop is not based on Eclipse so we cannot simply deploy Hamsters directly into it. The solution was to preserve the generated plugin from Eclipse which is basically an OSGi bundle. Than we created a new runtime environment for PetShop, without touching its internal modules, running it on top of an OSGi layer. Therefore, PetShop can know interrogate Hamsters using OSGi service calls (see figure 7.3 on the next page for an overview of the final architecture). In the same fashion, PetShop can be ran inside the OSGi layer of Hamsters standalone application. In addition, this approach enabled us to solve a conflict that has arisen when trying to integrate some task models with PetShop. This is especially true for the case of Amboss when the team wanted to integrate it into the PetShop CASE environment. The conflict was not very clear technically but it has its foundation in threads conflicts between basically AWT (PetShop is Swing based) and SWT. Although our tool uses EMP which is based on SWT, no conflicts arose because we are running both applications on top of the OSGi layer but with different runtime configurations.

[1] *http://www.eclipse.org/equinox/*

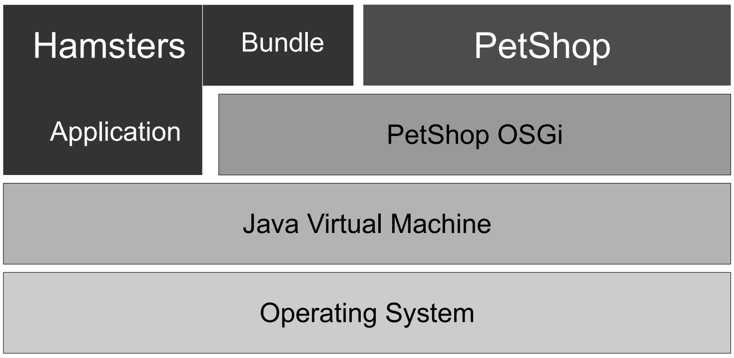


Figure 7.3: Hamsters and PetShop integration using OSGi

Conclusion and Prospects

Conclusion

This research topic helped us to draw some important conclusions. We learned that the activity of designing and developing software systems does not need to rely only on pure formal rigid approaches. The success of these systems depends largely not on their logical function but their efficiency in improving performance. Undoubtedly, creating *correct* systems is a requirement but limiting our focus to technical aspects will likely produce defect products. The software usage and its relationships with the external world are of an enormous importance too, underlining the signification of taking into consideration both social and technical aspects while designing any system.

Sometimes, we need to deal with *vague* concepts which are usually avoided by software engineers for their imprecision; this is typically true for social factors. However, we can always find a way to encode them with a minimum formality allowing us to put them into use. This is the case of Task Analysis in our research which was *formalized* into task models.

Developing Hamsters helped us to establish some good practices for defining models in general. Such activity should be carried in the same way as we wanted for system design. In other terms, developing a model is a design activity itself demanding to consider different socio-technical aspects behind it (in this case our target users are system designers). This is done through careful definition of the model from different perspectives by asking some basic questions:

- What are we modeling?
- Why we need this model?
- In which context the model will be used?
- What are the relevant real-world aspects this model needs?
- How should we represent this model (notation)?
- Does this model relate to other models? If yes in which way?
- How should *modelers* interact with the model tool (interaction and usability)?

Finally, this research allowed us to discover some insights from the Human-Computer Interaction discipline. Mainly, it enables the researcher to have a broader view beyond its domain and to look at other disciplines relating them to the problem in hand. Another conclusion which lies at the heart of our research is realizing the importance of task modeling in HCI and in software engineering in general. In particular, how HCI is crucial to software engineering if not a motor-factor.

Prospects

Hamsters is still in its early stages of research and development. This work provides a first version which will eventually evolve to support additional features and/or adapt to new findings by the Task Modeling community. Thus, its current form has various opportunities for enhancements. Primarily, finding solutions to the various criticisms of Hamsters discussed in our research text. As for the rest of this section, we will give some practical prospects that could be undertaken to improve our task model.

The first prospect we propose is developing a multi-diagram notation. Basically, we think of two diagrams of the same model–The first captures the structure, and the second expresses the flow. This approach helps us separate concerns on one hand and provide better notation on the other. The concern of the first diagram is static in nature and aims at providing a simple notation to describe the structure of our task model (only tasks are present). The second diagram is behavioral in nature and can resemble to some extend to a multi-level flow diagram: enabling us *a priori* to better describe flows (using existing flow diagrams notation for instance) and to enhance task flows readability (structure *is* of a second-concern here). By multi-level, we mean supporting some kind of semantic-zooming (e.g. the Fish-Eye zoom [Furnas 1986]) imporiving navigability from one abstraction level to another. This modification will affect only the notation level meaning that the model itself will not be affected. However, working on this prospect will need to demonstrate how such separation enhances the expressiveness and usability of our model. It should take into account some side-effects such as notation confusion (the same model is represented in two different diagrams).

The second prospect is aiming at increasing the formality of our model, we propose two formal languages:

Task Constraint Language A constraint based language that can be used mainly to describe conditions in a more formal way. This language should allow analysts to encode various types of conditions (basically on tasks, roles and objects). It should support logical operators such as *And*, *Or*, etc. It can be very useful to the simulation module as it permits better autonomy.

Task Query Language This language enables the user to query the task model for specific information. Basically, it should allow the analyst to carry projections on (1) tasks and (2) roles. The first helps the analyst filter/analyze tasks depending on various constraints (type, role, etc.). The second makes it easier to analyze roles and determine their tasks, detect clashes, assess task allocation... This language could be textual (resembling a simplified version of SQL) or graphical like the one used in Amboss [Giese et al. 2008].

However, these languages need to be very intuitive and simple to use as task analysts do not have necessarily a background in programming or formal languages.

Bibliography

An agenda for human-computer interaction: science and engineering serving human needs. *SIGCHI Bull.*, 23(4):17–32, 1991. ISSN 0736-6906.

OSGi Alliance, editor. *OSGi Service Platform, Core Specification, Release 4, Version 4.1.* aQute, 2007. ISBN 978-90-79350-01-8.

R. Anderson, J. Carroll, J. Grudin, J. McGrew, and D. Scapin. *Human-computer Interaction: Interact, '90*, chapter The oft missed step in the development of computer-human interfaces: Its desirable nature, value and role, pages 1051–1054. 1990.

J. Annett and K. Duncan. Task analysis and training design. *Occupational Psychology*, 41:211–227, 1967.

John Annett and Neville Stanton. *Task Analysis*, chapter Research and developments in task analysis, page 1. CRC Press, 2000.

M. Baron, V. Lucquiaud, D. Autard, and D. L. Scapin. K-made: un environnement pour le noyau du modèle de description de l'activité. In *IHM '06: Proceedings of the 18th International Conference of the Association Francophone d'Interaction Homme-Machine*, pages 287–288, New York, NY, USA, 2006. ACM. ISBN 1-59593-350-6.

Charles E. Billings. *Aviation Automation: The Search for a Human-Centered Approach.* Lawrence Erlbaum Associates, 1997.

Alan Blackwell and Thomas Green. *HCI models, theories, and frameworks: Toward a multidisciplinary science.*, chapter Notational Systems–The Cognitive Dimensions of Notations Framework., pages 103–133. Morgan Kaufmann Publishers, San Francisco, 2003.

B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(21):61–72, 1988.

Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987. ISSN 0169-7552.

Birgit Bomsdorf and Gerd Szwillus. From task to dialogue: task-based user interface design. *SIGCHI Bull.*, 30(4):40–42, 1998. ISSN 0736-6906.

Birgit Bomsdorf and Gerd Szwillus. Tool support for task-based user interface design. In *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*, pages 169–170, New York, NY, USA, 1999. ACM. ISBN 1-58113-158-5.

Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. ISSN 0018-9162.

Stuart Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human Computer Interaction.* Lawrence Erlbaum Associates, 1983.

John M. Carroll and Judith Reitman Olson, editors. *Mental models in human-computer interaction: research issues about what the user of software knows.* National Academy Press, Washington, DC, USA, 1987. ISBN 0-309-00266-0.

Alphonse. Chapanis. *Research techniques in human engineering.* Johns Hopkins University Press, 1959.

Alan Cooper. *The Inmates are Running the Asylum: Why High-tech Products Drive Us Crazy and How to Restore the Sanity.* Sams, 2 edition, March 1999. ISBN 0672326140.

K.J.W. CRAIK. Theory of the human operaor in control systems. i: The operator as an engineering system. *British Journal of Psychology*, 38:56–61, 1947.

K.J.W. CRAIK. Theory of the human operaor in control systems. ii: Man as an element in a control system. *British Journal of Psychology*, 38:142–148, 1948.

Johnson C.W. Proving properties of accidents. *Reliability Engineering and System Safety*, 67:175–191, 2000.

K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Computer Science Department, Technical University of Ilmenau, Ilmanau, Germany, 1998.

Paulo Pinheiro da Silva. User interface declarative models and development environments: A survey. In *Interactive Systems Design, Specification, and Verification*, volume 1946 of *Lecture Notes in Computer Science*, pages 207–226. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-41663-0. URL `http://www.springerlink.com/content/6q0n3xw31deutjac/`.

T. DeMarco. Structured analysis and system specification. pages 409–424, 1979.

D. Diaper. *Task analysis for human-computer interaction*, chapter Task Analysis for Knowledge Descriptions (TAKD): The method and an example, pages 108–159. Ellis Horwood, 1989.

D. Diaper. *People and Computers*, volume XIV, chapter Hardening Soft Systems Methodology, pages 183–204. Springer, 2000.

D. Diaper. The model matters: constructing and reasoning with heterarchical structural models. In G. Kadoda, editor, *Proceedings of the Psychology of Programming Interest Group 13th*, pages 191–206, 2001.

D. Diaper and P. Johnson. *Cognitive ergonomics and human-computer interaction*, chapter Task Analysis for Knowledge Descriptions: Theory and application in training, pages 191–224. Cambridge University Press, 1989.

Anke Dittmar, Peter Forbrig, Simone Heftberger, and Chris Stary. Support for task modeling - a "constructive" exploration. In *Engineering Human Computer Interaction and Interactive Systems*, volume 3425 of *Lecture Notes in Computer Science*, pages 59–76. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-26097-4. URL `http://www.springerlink.com/content/bvd9gg76mgly000b/`.

Ozeas V. Santana Filho and Konrad G. Kochan. The importance of requirements engineering for software quality. In *ISAS-SCI '01: Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics*, pages 529–532. IIIS, 2001. ISBN 980-07-7541-2.

G. W. Furnas. Generalized fisheye views. *SIGCHI Bull.*, 17(4):16–23, 1986. ISSN 0736-6906.

Susan Gasson. Human-centered vs. user-centered approaches. *Journal of Information Technology Theory and Application (JITTA)*, 5 (2):29–46, 2003. URL `http://hdl.handle.net/1860/1978`.

Matthias Giese, Tomasz Mistrzyk, Andreas Pfau, Gerd Szwillus, and Michael von Detten. AMBOSS: A task modeling approach for safety-critical systems. In *Engineering Interactive Systems*, volume 5247 of *Lecture Notes in Computer Science*, pages 98–109. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-85991-8.

Frank Bunker Gilbreth and Robert Thurston Kent. *Motion study: A method for increasing the efficiency of the workman.* D. Van Nostrand Company, 1911.

Fausto Giunchiglia, John Mylopoulos, and Anna Perini. The tropos software development methodology: processes, models and diagrams. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 35–36, New York, NY, USA, 2002. ACM. ISBN 1-58113-480-0.

J. Gould, S.J. Boies, and J. Ukelson. *Handbook of HumanComputer Interaction*, chapter How To Design Usable Systems, pages 231–254. Morgan Kaufmann Publishers Inc., 2 edition, 1997. ISBN 1-55860-246-1.

John D. Gould and Clayton Lewis. Designing for usability—key principles and what designers think. In *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 50–53, New York, NY, USA, 1983. ACM. ISBN 0-89791-121-0.

John D. Gould, Stephen J. Boies, and Clayton Lewis. Making usable, useful, productivity-enhancing computer applications. *Commun. ACM*, 34(1):74–85, 1991. ISSN 0001-0782.

T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

Jan Gulliksen, Bengt Göransson, Inger Boivie, Jenny Persson, Stefan Blomkvist, and Åsa Cajander. Key principles for user-centred systems design. In *Human-Centered Software Engineering - Integrating Usability in the Software Development Lifecycle*, volume 8 of *Human-Computer Interaction Series*, pages 17–36. Springer Netherlands, 2005. ISBN 978-1-4020-4027-6 (Print) 978-1-4020-4113-6 (Online). URL `http://www.springerlink.com/content/r340755323560170/`.

JoAnn T. Hackos and Janice C. Redish. *User and task analysis for interface design*. John Wiley & Sons, Inc., New York, NY, USA, 1998. ISBN 0-471-17831-4.

Erik Hollnagel. *Barriers and accident prevention*. Ashgate Publishing, 2004.

ISO/IEC. *ISO 13407:1999 Human-centred design processes for interactive systems*. ISO, Geneva, Switzerland, 1999.

David H. Jonassen, Martin Tessmer, and Wallace H. Hannum. *Task analysis methods for instructional design*. Lawrence Erlbaum Associates, 1999.

D.G. Jones and M.R. Endsley. Overcoming representational errors in complex environments. *Human Factors*, 42 (3):367–378, 2000. URL `http://hfs.sagepub.com/cgi/content/short/42/3/367`.

Mitchell Kapor. A software design manifesto. *Dr. Dobb's J.*, 16(1):62–67, 1991. ISSN 1044-789X.

David Kieras. Goms modeling of user interfaces using ngomsl. In *CHI '94: Conference companion on Human factors in computing systems*, pages 371–372, New York, NY, USA, 1994. ACM. ISBN 0-89791-651-4.

R. Kimball. Is er modeling hazardous to dss? *DBMS Magazine*, 1995.

B. Kirwan and L. K. Ainsworth, editors. *A Guide to Task Analysis: The Task Analysis Working Group*. Taylor & Francis, 1 edition, September 1992. ISBN 0748400583.

Linda T. Kohn, Janet M. Corrigan, and Molla S. Donaldson, editors. *To Err Is Human: Building a Safer Health System*. National Academic Press, 2000.

Morten Kyng. Making representations work. *Commun. ACM*, 38(9):46–55, 1995. ISSN 0001-0782.

Benjamin A. Lieberman. *The Art of Software Modeling*. Auerbach Publications, Boston, MA, USA, 2006. ISBN 1420044621.

Quentin Limbourg and Jean Vanderdonckt. Comparing task models for user interface design. 2003.

V. Lucquiaud. Proposition d'un noyau et d'une structure pour les modèles de tâches orientés utilisateurs. In *ConfÃ©rence Francophone sur l'Interaction Homme-Machine*, pages 83–90, Toulouse, France, 2005.

T. Mandel. *The Elements of User Interface Design.* Wiley Computer Publishing, New York, 1997.

MetaCase. Abc to metacase technology (white paper). Technical report, MetaCase Inc., 2004.

George Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.

Daniel Moody. What makes a good diagram? improving the cognitive effectiveness of diagrams in is development. In Knapp and Magyar, editors, *Intl Conf on Information Systems Development*. Springer, August 31-2 2006.

J. Morgan and P. Welton. *See What I Mean?* Edward Arnold, London, 1992.

D.C. Nagel. *Human factors in aviation*, chapter Human error in aviation operations, pages 263–303. Academic Press, 1988.

David Navarre, Philippe Palanque, and Marco Winckler. Task models and system models as a bridge between hci and software engineering. In *Human-Centered Software Engineering*, Human-Computer Interaction Series, pages 357–385. Springer London, 2009. ISBN 978-1-84800-906-6 (Print) 978-1-84800-907-3 (Online). URL `http://www.springerlink.com/content/m98vrr1216p40g83/`.

Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993.

Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256, New York, NY, USA, 1990. ACM. ISBN 0-201-50932-6.

J. C. Nordbotten and M. E. Crosby. The effect of graphic style on data model interpretation. *Information Systems Journal*, 9(2):139–155, 1999.

Donald A. Norman. *The Design of Everyday Things.* Basic Books, September 2002. ISBN 0465067107.

Donald A. Norman and Stephen W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1986. ISBN 0898597811.

Nadine Ozkan, Cecile Paris, and Bill Simpson-Young. Towards an approach for novel design. *Computer-Human Interaction, Australasian Conference on*, 0: 186, 1998.

Philippe Palanque and Rémi Bastide. *Critical Issues in User Interface Systems Engineering*, chapter Task Models - System Models: a Formal Bridge Over the Gap, pages 65–79. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 3540199640.

R. Parasuraman, R. Molloy, M. Mouloua, and B. Hilburn. *Automation and human performance*, chapter Automation and human performance: Theory and applications, pages 91–115. Lawrence Erlbaum Associates, 1996.

Fabio Paternò. Model-based design of interactive applications. *Intelligence*, 11 (4):26–38, 2000. ISSN 1523-8822.

Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. Concurtasktrees: A diagrammatic notation for specifying task models. In *INTERACT '97: Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd. ISBN 0-412-80950-8.

Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995. ISSN 0001-0782.

J. Preece, Y. Rogers, H. Sharp, D. Helen, Benyon, S. Holland, and T. Carey. *Human-Computer Interaction*. Addison Wesley, 1994.

Carol Righi and Janice James. *User-Centered Design Stories: Real-World UCD Case Studies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123706084, 9780123706089.

M.S. Sanders and E.J. McCormick. *Human factors in engineering and design*. McGraw-Hill, 7 edition, 1992.

DL. Scapin and C. Pierret-Golbreich. Une méthode analytique de description des tâches. *Colloque sur l'ingénierie des Interfaces Homme-Machine*, pages 131–148, 1989.

Wilbur Lang Schramm. *The Process and Effects of Mass Communication*. University of Illinois Press, revised edition (october 1971) edition, 1971.

G.A. Sexton. *Human factors in aviation*, chapter Cockpit-crew systems design and integration, pages 495–526. Academic Press, 1988.

Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2 edition, March 2007. ISBN 0470018666.

A. Shepherd. *Hierarchical Task Analysis*. Taylor and Frances, 2001.

Bruce Tognazzini. *Tog on software design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996. ISBN 0-201-48917-1.

UPA. What is user-centered design?, June 2009. URL `http://www.usabilityprofessionals.org/`.

Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10: 2002, 2001.

& Butler M. B. Vredenburg, K. Current practice and future directions in user-centered design. In *Proceedings of the Usability Professionals' Association Fifth Annual Conference*, 1996.

Karel Vredenburg, Ji-Ye Mao, Paul W. Smith, and Tom Carey. A survey of user-centered design practice. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 471–478, New York, NY, USA, 2002. ACM. ISBN 1-58113-453-3.

Cathleen Wharton, John Rieman, Clayton Lewis, and Peter Polson. The cognitive walkthrough method: a practitioner's guide. pages 105–140, 1994.

Terry Winograd, John Bennett, and Laura De Young. *Bringing Design to Software*. Addison-Wesley Professional, April 1996. ISBN 0201854910.